Jürgen Kotz

erfolgreich Visual Basic 2010 programmieren

ADDISON-WESLEY

Visual Basic 2010

Jürgen Kotz

erfolgreich Visual Basic 2010 programmieren

ADDISON-WESLEY

An imprint of Pearson Education

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

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

10 9 8 7 6 5 4 3 2 1

13 12 11

ISBN 978-3-8273-2951-6

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Lektorat: Brigitte Bauer-Schiewek, bbauer@pearson.de
Fachlektorat: Walter Saumweber
Korrektorat: Petra Kienle
Herstellung: Martha Kürzl-Harrison, mkuerzl@pearson.de
Coverkonzeption und -gestaltung: Thomas Arlt (tarlt@adesso21.net)
Satz: Reemers Publishing Services GmbH, Krefeld (www.reemers.de)
Druck und Verarbeitung: Kösel, Krugzell (www.KoeselBuch.de)

Printed in Germany

Auf einen Blick

Inhaltsverzeichnis

Vorwort

Seit einigen Monaten ist nun die endgültige Version von Visual Studio 2010, das sich in einem neuen Kleid zeigt, und des .NET Framework 4.0 auf dem Markt. Microsoft hat dabei bewiesen, dass vor allem durch die Erhöhung der Produktivität für den Entwickler und einige angenehme neue Features in der Entwicklungsumgebung auf die Wünsche der Entwickler eingegangen wurde. Die Einführung von dynamischen Sprachen, der Task Parallel Library und dem Managed Extensibility Framework sind die großen neuen Themen der aktuellen Version. Die vorliegende Version von Visual Basic wird sowohl als Visual Basic 10 als auch als Visual Basic 2010 bezeichnet.

Ich hoffe, Ihnen mit diesem Buch den Umstieg oder auch den Einstieg so einfach wie möglich zu gestalten.

Viele Bücher zu diesem Thema sprechen direkt Einsteiger an, die noch nie mit einer Programmiersprache gearbeitet haben. Die meisten Einsteiger haben aber zumindest ein grundlegendes Wissen über andere Programmiersprachen und möchten direkt in das Geschehen einsteigen und ihr vorhandenes Wissen schnell erweitern, anstatt 50 Seiten über den Aufbau der Entwicklungsumgebung zu lesen.

Um diesem Anspruch gerecht zu werden, habe ich mich immer wieder gefragt, was für Sie als Leser besonders von Bedeutung ist. Herausgekommen ist dieses Werk, das weiterführende Programmiertechniken und praxisnahe Beispiele beinhaltet, mit denen es auch möglich ist, größere Projekte zu planen und zu realisieren.

Für Fragen, Kritik und Anregungen stehe ich Ihnen selbstverständlich jederzeit zur Verfügung.

Sie erreichen mich unter:

juergen@primetime-software.de

Kapitelübersicht

Kapitel 1: Das erste Kapitel liefert aktuelle Informationen zum .NET Framework 4.0. Sie als Leser bekommen dabei einen schnellen Einstieg in die Welt der .NET-Programmierung.

Kapitel 2: Das zweite Kapitel beschäftigt sich mit der Entwicklungsumgebung Visual Studio 2010. Alle Neuerungen sowie altbekannte und verbesserte Module werden erklärt, damit Sie sich als VB.NET- oder VB 6-Entwickler schnell zurechtfinden.

Kapitel 3: Dieses Kapitel wiederholt kurz die elementaren Begriffe der Visual Basic 10-Programmierung, um Ihr Wissen aufzufrischen und auf den neuesten Stand zu bringen.

Kapitel 4: In diesem Kapitel wird auf das Exceptionhandling oder, wie es so schön heißt, auf die strukturierte Fehlerbehandlung unter .NET eingegangen.

Kapitel 5: In diesem Kapitel wird auf die objektorientierten Fähigkeiten von Visual Basic 10 eingegangen. Hier lernen Sie das tägliche Werkzeug in Bezug auf Klassen, Vererbung, Interfaces und Polymorphie kennen.

Kapitel 6: In diesem Kapitel werden die neuesten Features von Visual Basic 10 vorgestellt. Die neuen Sprachfeatures wie dynamische Programmierung und eine kurze Einführung in das Managed Extensibility Framework sind die zentralen Themen in diesem Abschnitt.

Kapitel 7: Im siebten Kapitel erfahren Sie einiges über Visual Basic 10 in puncto Windows Forms. Lernen Sie neben den Steuerelementen die Erstellung eigener Steuerelemente sowie den flexiblen Umgang mit Ressourcendateien und der Lokalisierung Ihrer Anwendung in verschiedenen Sprachen.

Kapitel 8: Das achte Kapitel gibt einen kurzen Ausblick auf die Alternative zu Windows Forms – WPF. Die Windows Presentation Foundation basiert auf einem komplett anderen Konzept zur Darstellung der GUI. Und dies wird hier kurz vorgestellt.

Kapitel 9: Das neunte Kapitel beschäftigt sich mit einem der wichtigsten Themen überhaupt: ADO.NET. Fast kein Entwickler kommt um eine Datenbank für die Speicherung der bearbeiteten Daten herum. In diesem groß angelegten Kapitel lernen Sie, wie Sie eine Datenbankanwendung planen und aufbauen und was Sie dabei alles beachten sollten. Ebenfalls erfahren Sie mehr über das immer wichtiger werdende Thema XML mit praktischen Codebeispielen.

Kapitel 10: Lassen Sie sich mit diesem Kapitel in die neue Welt von LINQ entführen und betrachten Sie Beispiele für LINQ to Objects, LINQ to SQL und LINQ to XML und natürlich LINQ to Entities.

Kapitel 11: ClickOnce Deployment ist der Schwerpunkt dieses Kapitels. Was kann ClickOnce Deployment alles und welche Optionen haben Sie dabei? Außerdem wird auch noch die Softwareverteilung mittels Windows Installer kurz vorgestellt.

Kapitel 12: Basisklassen machen die effiziente Arbeit mit Visual Basic 10 erst möglich. Lernen Sie, wie Sie Zugriff zu Daten auf Ihrer Festplatte bekommen und wie Sie Dateien und Ordner mit Visual Basic 10 ansteuern können. Drawing, Serialisierung, Multithreading und Parallelisierung sowie Drucken sind weitere Themen des Kapitels.

Kapitel 13: Interoperabilität – wie kann ich Win32-API-Funktionen aufrufen und wie funktioniert die Kommunikation mit COM-Komponenten?

Kapitel 14: Und zum Schluss gehen wir noch über zur Webentwicklung und wollen die interessantesten und neuen Features von ASP.NET 4.0 vorstellen.

Achtung

Aufgrund von unterschiedlichen Hardware- und Softwareinstallationen können wir leider nicht garantieren, dass alle vorgestellten Programme oder Programmteile sofort auf Anhieb funktionieren. Wir können aus denselben Gründen auch keine Haftung für irgendwelche Folgen übernehmen, die sich aus dem Benutzen des hier und auf der CD veröffentlichten Codes ergeben. Wir sind uns trotzdem ziemlich sicher, dass die vorgestellten Programme auch bei Ihnen ohne großen Aufwand lauffähig sein werden.

Danke

Zum Schluss will ich mich noch bei allen recht herzlich bedanken, die bei der Erstellung dieses Buchs mitgeholfen haben.

Einen herzlichen Dank an meine Lektorinnen Frau Sylvia Hasselbach und Frau Brigitte Bauer-Schiewek für die Unterstützung während des Projekts sowie an Herrn Walter Saumweber für die hervorragende Zusammenarbeit beim Fachlektorat. Herr Saumweber hat mit seinem fundierten Wissen für die Sicherstellung der hohen Qualität dieses Buchs gesorgt. Ein Dank auch an Frau Petra Kienle für das gewissenhafte Dudenlektorat sowie allen anderen fleißigen Hände bei Addison-Wesley.

Ebenso vielen Dank an den ASP.NET-MVP Christian Wenz für die Erstellung beim ASP.NET-Kapitel.

1 .NET Framework

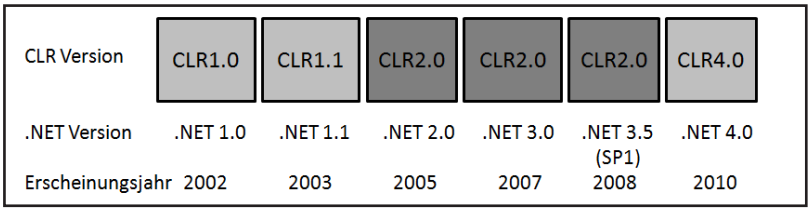
Das .NET Framework ist der wichtigste Bestandteil der Microsoft .NET-Strategie. Es handelt sich hierbei um eine umfangreiche Programmierumgebung, die mit verschiedenen Programmiersprachen und Programmen eine neue Generation der Programmierung einläutete. Das .NET Framework bietet dabei auch eine Laufzeitumgebung für .NET-Anwendungen, die sogenannte Common Language Runtime. Mit dem .NET Framework können Sie in unterschiedlichen Programmiersprachen umfangreiche Windows- und Webapplikationen entwickeln und bereitstellen. Hierzu enthält das .NET Framework sehr viele Klassen, mit denen es möglich ist, alle erdenklichen Funktionen (wie zum Beispiel den Zugriff auf eine Datenbank) auszuüben. Dieses Kapitel behandelt die grundlegenden Bestandteile des Microsoft .NET Framework.

1.1 Geschichte

Zunächst verfolgte Microsoft den Plan, eine neue Version des *Component Object Model* (COM) zu entwickeln und zu veröffentlichen. Doch schon bald stellte sich heraus, dass sich die Neuentwicklung immer weiter von COM entfernte. Microsoft stellte im Juli 2000 das .NET Framework das erste Mal der Öffentlichkeit vor. Mit dem .NET Framework sollte es möglich werden, eine Basis zur Ausführung und Anwendung von Programmen zur Verfügung zu stellen. Mit der *Common Language Infrastructure* (CLI) sollte es ab nun egal sein, mit welcher Programmiersprache man seine Anwendungen programmiert. Für die Ausführung der Programme war die Common Language Runtime (CLR) verantwortlich. Im Januar 2002 veröffentlichte Microsoft das .NET Framework 1.0 zusammen mit Visual Studio 2002. Im Jahre 2003 veröffentlichte Microsoft mit Visual Studio 2003 das .NET Framework 1.1. Dieses beinhaltete einige Verbesserungen zum Beispiel im Bereich der Security. Ende des Jahres 2005 erschien dann die nächste Version 2.0 mit dem neuen Visual Studio 2005. Innerhalb von 2.0 wurden nochmals viele weitere Klassen dem Framework hinzugefügt, um das Arbeiten noch effektiver zu gestalten. Generische Klassen und wesentliche Verbesserungen in ASP.NET 2.0 und ADO.NET 2.0 waren die Highlights dieser Version. Das Framework 2.0 brachte auch eine

neue Version 2.0 der Common Language Runtime. Im Jahr 2007 erschien dann das Framework 3.0, dieses Mal ohne eine neue Visual Studio-Version. Das .NET Framework 3.0 erweiterte dabei die Vorgängerversionen um die Windows Presentation Foundation (WPF), die Windows Communication Foundation (WCF) sowie die Windows Workflow Foundation (WF). Für die Ausführung von .NET Framework 3.0-Programmen war weiterhin die CLR 2.0 zuständig. Das Gleiche galt später auch für Programme, die auf der Version 3.5 des .NET Framework basierten. Diese Version erschien zusammen mit Visual Studio 2008 ein Jahr nach der Version 3.0. 2010 erschien dann die momentan aktuellste Version 4.0 des Framework mit der neuen Entwicklungsumgebung Visual Studio 2010. Mit der Einführung von 4.0 erschien auch eine neue Runtime, die CLR 4.0. Abbildung 1.1 demonstriert die Entwicklung des .NET Framework.

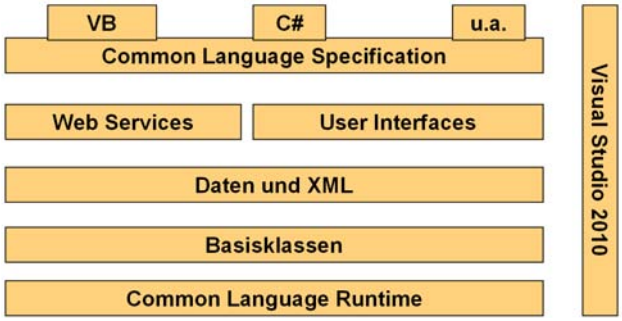
Abbildung 1.1
Entwicklung des .NET Framework



1.2 Aufbau des .NET Framework

Den Aufbau des .NET Framework sehen Sie in Abbildung 1.2.

Abbildung 1.2
Der Aufbau des .NET Framework im Überblick



1.2.1 Die Common Language Runtime

Die *Common Language Runtime* (kurz CLR) stellt die Laufzeitumgebung für die .NET-Programmiersprachen dar und ist somit das Herz des .NET Framework. Früher war für die Ausführung Ihres Codes das Betriebssystem selbst verantwortlich. Diese Aufgabe übernimmt seit der Einführung von .NET die Common Language Runtime. Hierbei ist es gleichgültig, ob der Code der Anwendung in Visual Basic, C# oder einer anderen Programmiersprache des .NET Framework erstellt worden ist.

Abbildung 1.3 zeigt die Bestandteile der Common Language Runtime.

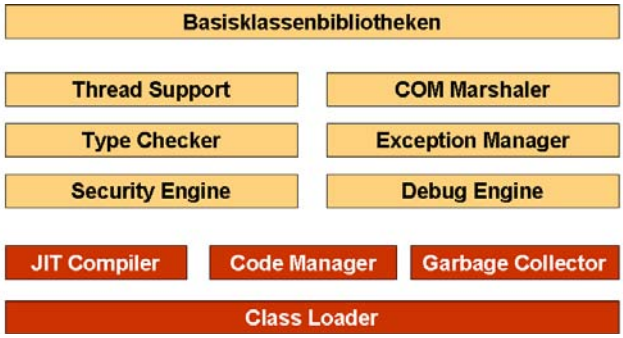


Abbildung 1.3
.NET Framework im Detail

Die *Microsoft Intermediate Language* (MSIL) sorgt dafür, dass der Code, solange er in einer Programmiersprache des .NET Framework erstellt worden ist, ausgeführt werden kann. Egal, in welcher Programmiersprache Sie Ihre Anwendung schreiben, beim Kompilieren wird MSIL erzeugt.

Microsoft Intermediate Language mag einigen besser unter dem Namen *managed Code* oder zu Deutsch verwalteter Code bekannt sein. Die vom Visual Studio erstellten Exe- und DLL-Dateien (im .NET-Jargon spricht man von **Assemblies**) enthalten lediglich den MSIL-Code, eine Art Zwischencode ohne maschinen- und plattformabhängige Befehle. Dieser ist nicht direkt vom Prozessor ausführbar.

.NET kompiliert den benötigten MSIL-Code bei der Ausführung (Just in Time) durch die CLR in Maschinencode. Diese Methode, die den Code erst bei Bedarf kompiliert, nennt man auch Just-in-time-Kompilation, die der Just-in-Time-Compiler (JIT oder kurz Jitter) übernimmt. Die Arbeitsweise des Just-in-Time-Compilers ist in Abbildung 1.4 illustriert.

Im Vergleich zu »alten« Visual-Basic-Versionen ergeben sich daraus zwei Vorteile. Zum einen wurden alte Visual-Basic-Programme interpretiert und nicht kompiliert. Bei der Kompilierung wird die Übersetzung in Maschinencode ein einziges Mal durchgeführt. Ein Interpreter dagegen übersetzt jede Anweisung immer wieder aufs Neue, was vor allem bei Schleifen einen wesentlichen Geschwindigkeitsnachteil mit sich bringt.

Visual Basic 6 kompilierte schon beim Erstellen des Programms. Damit waren jedoch im übersetzten Code bereits maschinen- und prozessorabhängige Informationen enthalten, die eine Portierung auf andere Plattformen unmöglich machten.

Unter .NET haben wir jetzt eine Kompilierung zur Laufzeit, die einen Geschwindigkeitsvorteil gegenüber dem Interpretersystem bietet und trotzdem Plattformunabhängigkeit bereitstellt.

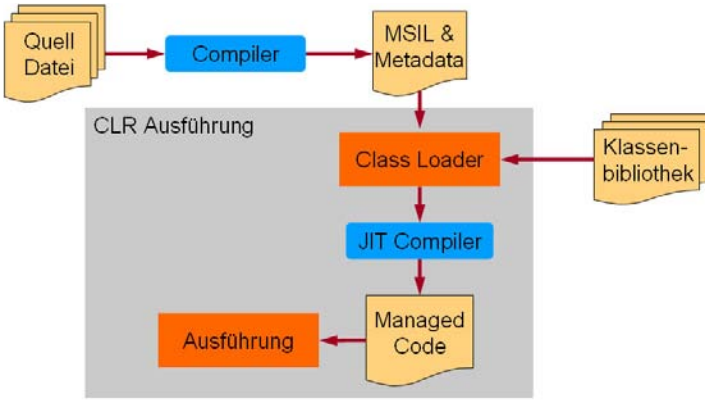


Abbildung 1.4
Arbeitsweise des Just-in-Time-Compilers

1.3 Basisklassenbibliothek

Eines der wichtigsten Elemente im .NET Framework stellt die Basisklassenbibliothek dar. Diese Bibliothek bietet Zugriff auf Tausende Funktionen, die im System vorhanden sind. Die Klassen ermöglichen den Zugriff beispielsweise auf Datenbanken, Systemfunktionen, Internet- und Netzwerkfunktionen. Sie bieten eine, auf den ersten Blick, fast undurchschaubare Masse an Möglichkeiten für den Programmierer.

Eine Klasse stellt dabei verschiedene Funktionalitäten von Diensten zur Verfügung, die ein Programmierer ansonsten selbst schreiben müsste. Klassen bestehen vornehmlich aus Funktionen, Prozeduren und Eigenschaften.

Falls Sie mit Visual Basic 6 programmiert haben, ist Ihnen der Begriff API sicherlich bekannt. Mit dem Application Programming Interface lieferte Microsoft eine wichtige Schnittstelle zum Windows-Betriebssystem. Mit dem .NET Framework wurden APIs komplett über Bord geworfen und durch Klassenbibliotheken ersetzt.

Falls Ihnen die Vielzahl von Klassen einmal nicht reichen sollte, können Sie auch zu jeder Zeit eine eigene Klasse schreiben und in Ihre Entwicklung mit einfließen lassen. Wie bei der CLR ist es bei den Klassenbibliotheken ebenfalls egal, in welcher Programmiersprache die Klasse erstellt worden ist. Das bedeutet, Sie können eine Klasse, die in Visual Basic erstellt worden ist, in C# verwenden und umgekehrt.

Einen Auszug aus den Basisklassen-Namensräumen sehen Sie in Abbildung 1.5.

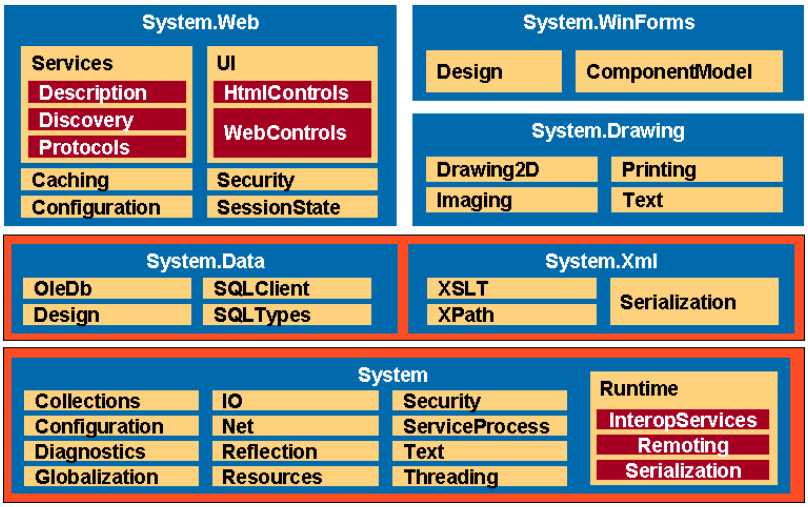


Abbildung 1.5
Basisklassenhierarchie des .NET Framework

1.3.1 Common Type System

Das *Common Type System* (CTS) definiert, wie verschiedene Typen deklariert, benutzt und während der Laufzeit verwaltet werden sollen. Das Common Type System stellt unter anderem eine Richtlinie dar, wenn Ihr Projekt mit verschiedenen .NET-Programmiersprachen arbeitet. Hierbei wird sichergestellt, dass beispielsweise Variablen typsicher sind und dass der erzeugte Code möglichst schnell ausgeführt werden kann.

Sämtliche Datentypdefinitionen liegen im Framework und nicht mehr innerhalb der verschiedenen Sprachen. Somit ist sichergestellt, dass die Datentypen in allen .NET-Programmiersprachen identisch sind.

Info

An diesem Punkt möchte ich nur kurz an die Problematik der Stringverarbeitung erinnern, wenn Sie mit einem VB6-Programm über eine API eine C-Funktion aufgerufen haben, die als Parameter einen String erwartete.

1.3.2 Common Language Specification

Die *Common Language Specification* (CLS) ist eine Sammlung von grundlegenden Sprachfeatures und Richtlinien, die für Ihre .NET-Anwendungen benötigt werden. Um sicherzugehen, dass auch die Kompatibilität der Typen unterschiedlicher .NET-Programmiersprachen untereinander gewährleistet ist, umfasst die CLS Richtlinien, die für den Entwickler in Form von Gruppen und Features zur Verfügung stehen. Mit dem Common Type System ist es möglich, Software, die in verschiedenen .NET-Sprachen geschrieben worden ist, zusammenarbeiten zu lassen.

Eine Sprache, die sich an diese Spezifikation hält, gilt als .NET-Programmiersprache.

Wenn Sie in Fachzeitschriften blättern oder sich mit anderen .NET-Entwicklern unterhalten, wird schnell das Wort managed Code fallen. Doch was verbirgt sich eigentlich hinter diesem managed Code?

Managed Code wird während der Ausführung durch die .NET Common Language Runtime verwaltet. Sie als Entwickler können so alle Vorteile nutzen, welche die Common Language Runtime bereitstellt, wie zum Beispiel die automatische Speicherbehandlung oder auch die Fehlerbehandlung.

1.3.3 WinForms und Webanwendungen

Neben Windows-Anwendungen, den sogenannten WinForms oder alternativ seit der Einführung des .NET Framework 3.0 die Windows Presentation Foundation (WPF), bietet das Microsoft .NET Framework auch die Möglichkeit, WebForms mit ASP.NET zu erstellen. ASP.NET ist der Nachfolger von Active Server Pages und bietet die Möglichkeit, Internetapplikationen zu erstellen.

Mit WinForms oder der WPF können Sie klassische grafische Windows-Anwendungen mit Formularen, Textfeldern, Buttons und so weiter erstellen. In diesem Buch werden wir uns fast ausschließlich dem Thema der Windows-Applikationen widmen.

1.3.4 Assemblies

Eine Assembly ist, einfach gesagt, eine elementare Komponente, die von einer Anwendung benutzt wird. Die Assemblies liegen im *bin*-Verzeichnis Ihrer Anwendung in Form von *.dlls* oder *.exe*-Dateien vor. Gerade durch Assemblies ist es nicht mehr nötig, Anwendungen zu installieren, wie Sie es aus Visual Basic 6-Zeiten gewohnt waren. Eine Assembly besteht aus insgesamt vier Elementen. Neben dem Microsoft Intermediate Language-Code (MSIL) enthält die Assembly Typmetadaten, Gruppen von verwendeten Ressourcen und das Assembly-Manifest mit folgenden Metadaten:

- Den Namen der Assembly
- Vierstellige Versionsnummer
- Optionale kulturbezogene Informationen (Sprache)
- Verweise zu den verschiedenen Ressourcen und Typen
- Abhängigkeiten zu eventuell anderen vorhandenen Assemblies
- Liste mit den Dateien in der Assembly

Die Assembly-Datei wird automatisch mit dem Namen *AssemblyInfo.vb* erzeugt. Diese kann über den Projektmappen-Explorer, wie alle anderen Dateien des Projekts, aufgerufen werden, wie Sie in Listing 1.1 sehen.

Listing 1.1 AssemblyInfo.vb einer Applikation

```
Imports System.Reflection
Imports System.Runtime.CompilerServices
Imports System.Runtime.InteropServices

' Allgemeine Informationen über eine Assembly werden über die
' folgende Attributgruppe gesteuert.
```

Listing 1.1 (Forts.) AssemblyInfo.vb einer Applikation

```
' Ändern Sie diese Attributwerte, um die
' AssemblyInFormationen zu ändern

' ToDo: Werte der Assembly-Attribute überprüfen
<Assembly: AssemblyTitle("")>
<Assembly: AssemblyDescription("")>
<Assembly: AssemblyCompany("")>
<Assembly: AssemblyProduct("")>
<Assembly: AssemblyCopyright("")>
<Assembly: AssemblyTrademark("")>
<Assembly: AssemblyCulture("")>

' Versionsinformationen für eine Assembly bestehen
' aus den folgenden vier Werten:
'Hauptversion
'Nebenversion
'Revision
'Buildnummer

' Sie können alle Werte festlegen oder für Revision und
' Buildnummer den Standard
' mit '*' verwenden. Siehe unten

<Assembly: AssemblyVersion("1.0.*")>
```

Private und globale Assemblies

Man unterscheidet zwischen privaten und globalen Assemblies. Eine private Assembly wird nur von einer einzigen Anwendung benutzt und liegt im gleichen Anwendungsverzeichnis wie die Anwendung (zugehörige Exe-Datei) selbst. Falls mehrere Assemblies in einem Verzeichnis liegen, müssen diese zueinander kompatibel sein. Die Installation kann dabei über die einfache XCopy-Methode erfolgen und folgt dem **NutShell**-Prinzip. Das heißt, sie hat keine Auswirkung auf die Laufweise von anderen installierten Anwendungen, sondern bleibt von diesen vollkommen unabhängig und getrennt.

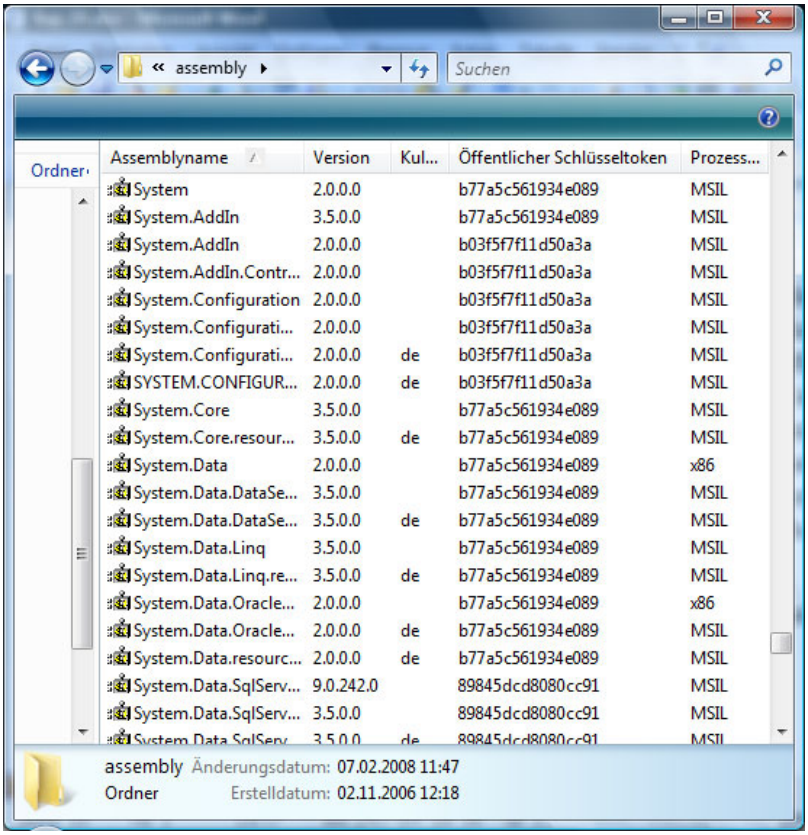
Globale oder auch Shared Assemblies können von beliebig vielen Anwendungen benutzt werden. Sie liegen im sogenannten Global Assembly Cache (GAC), einer eindeutigen Verzeichnisstruktur im Windows-Verzeichnis. Alle globalen Assemblies sind mit einem digitalen Schlüssel signiert, dieser ermöglicht ihre eindeutige Zuweisung. Anders als bei nicht signierten Assemblies erfolgt eine Versionsüberprüfung durch die Common Language Runtime. Eine Installation einer neuen Version einer Assembly ist dabei eine Parallelinstallation, die alte Version wird nicht überschrieben. Auf diese Art und Weise werden Kompatibilitätsprobleme, die unter COM als DLL-Hell bekannt wurden, verhindert.

Falls Ihre Assembly auch von mehreren Anwendungen benutzt werden soll, empfiehlt es sich, selbst eine globale Assembly anzulegen. Für die Erstellung eines Schlüssels für die digitale Signatur stehen Tools sowohl in Visual Studio wie auch ein Befehlszeilentool (*sn.exe*) zur Verfügung. Die Befehlszeilentools können Sie über die .NET-Eingabeaufforderung aufrufen. Die Eingabeaufforderung finden Sie im Startmenü-Programmverzeichnis unter dem Menüpunkt Microsoft Visual Studio 2010/Visual Studio Tools.

Um eine Assembly im globalem Assembly Cache anzulegen, dient der Befehl *GacUtil*. Mit *GacUtil-i NameDerAssembly* können Sie Ihre eigene Assembly im Cache ablegen. Der Parameter *i* steht hierbei für *install*. Eine Alternative ist, die Datei einfach per *Drag&Drop* im Explorer in den Global Assembly Cache zu ziehen (Kopieren und Einfügen hingegen ist nicht möglich).

Abbildung 1.6 zeigt den Inhalt des Global Assembly Cache auf meinem Rechner.

Abbildung 1.6
Das Windows-Assembly-Verzeichnis



Ladevorgang einer Assembly

Um zu verstehen, wie Assemblies wirklich arbeiten, ist es notwendig, dass man ihre genaue Arbeitsweise kennt. Beim Starten einer Anwendung führt die Runtime eine festgelegte Abfolge von Suchvorgängen aus. Zunächst wird das Assembly-Manifest ausgelesen. Falls die Assembly lokal im Applikationsverzeichnis vorliegt, es sich somit um eine private Assembly handelt, wird diese Assembly geladen. Eine Versionsüberprüfung findet hierbei nicht statt, außer die Assembly ist digital signiert. Falls es sich um eine globale Assembly handelt, werden die Konfigurationsdateien gelesen. Danach wird die gewünschte Version der Assembly im globalen Assembly Cache gesucht und geladen. Dieser Vorgang wird auch Probing genannt.

1.3.5 Sicherheit

Microsoft setzt das Sicherheitskonzept als eines der wichtigsten Elemente des .NET Framework fest. Enthalten sind Sicherheitsmechanismen zum Schutz von Programmen, beispielsweise vor Viren. Neben codebasierter Sicherheit (Code based Security) bietet das Framework auch noch das nutzerbasierte Modell (Role based Security).

Sicherheitsmodelle im Detail

Das .NET Framework bietet sechs Bereiche, um Sicherheit in Ihren eigenen Anwendungen zu implementieren. Hierzu zählen:

- Typsicherheit
- Code-Signatur (Strong Names)
- Kryptografie
- Rollenbasierte Sicherheit
- Code Access Security
- Isolated Storage

Typsicherheit

Unter Typsicherheit versteht man, dass Methoden nur über fest definierte Schnittstellen und passende .NET-konforme Parameter aufgerufen werden können. Während der Just-in-Time-Kompilierung wird der managed Code auf Typsicherheit geprüft. Dieses verhindert das Risiko von Codeausführungen oder provozierten unerwarteten Aktionen. Provozierte Ausnahmen können beispielsweise dazu verwendet werden, um mit falschen Werten Benutzerabfragen zu umgehen. Durch die Typensicherheit können Buffer Overruns in Ihrem eigenen Programm verhindert werden.

Tipp

Buffer Overruns oder Pufferüberläufe treten auf, wenn an Methoden Parameter übergeben werden, die größer als vorgesehen sind, und somit andere Speicheradressen (zum Beispiel Rücksprungadressen) überschrieben werden.

Die Typsicherheit spielt eine zentrale Rolle bei der Isolation von Assemblies. Typsicherer Code hat zur Folge, dass sich Assemblies untereinander nicht in ihrer Ausführung beeinträchtigen, und erhöht somit die Zuverlässigkeit und Lauffähigkeit Ihrer Anwendungen. Typsicherheit wird standardmäßig in Ihrer Anwendung verwendet. Falls Sie keinen managed Code geschrieben haben, wird eine Wahrheitsnachweis-Ausnahme ausgeführt.

Code-Signierung

Die Code-Signierung identifiziert Ihren Code eindeutig und sorgt dafür, dass dieser nicht nachträglich manipuliert werden kann. Eine Evidence enthält Informationen über die Identität sowie Herkunft einer benutzten Assembly. Anhand einer digitalen Signatur können durch das .NET Framework-Sicherheitssystem Berechtigungen zum Zugriff auf schützenswerte Ressourcen

(Dateisystem, Registry etc.) verteilt und festgelegt werden. Je mehr Beweise für die Herkunft erbracht werden, desto mehr Berechtigungen können der Assembly eingeräumt werden.

Kryptografie

Gerade bei sensiblen Daten Ihres Programms wie zum Beispiel Passwörtern ist es von großer Bedeutung, diese so gut wie möglich zu verschlüsseln, um sie vor Fremdzugriff zu schützen. Doch nicht nur bei Passwörtern und anderen Benutzerdaten kommt Kryptografie ins Spiel. Auch Daten, die über das Netzwerk oder über das Internet verschickt werden, sollten immer verschlüsselt gesendet werden, so dass gar nicht erst die Möglichkeit besteht, dass sie von einem unbefugten Dritten eingesehen werden können. Kryptografie ist im eigentlichen Sinne die Bezeichnung für Verschlüsselung (Chiffrierung) und Entschlüsselung (Dechiffrierung). Das .NET Framework bietet einen eigenen Namespace namens *System.Security.Cryptography*, der viele Klassen und Lösungswege enthält, um Kryptografie in die eigene Anwendung zu integrieren.

Bei der Kryptografie gibt es drei wichtige Gesichtspunkte. Zum einen wird die Datenintegrität über Hash-Algorithmen gewährleistet. Als Zweites steht die Authentifizierung über Signaturen zur Verfügung. Als Letztes bietet sie Ihnen als Entwickler noch die Verschlüsselung über symmetrische und asymmetrische Algorithmen.

Info

Eine Hash-Funktion (Streuwertfunktion) ist eine nicht umkehrbare Funktion (ein Algorithmus), die eine umfangreiche Menge an Daten in natürlichen Zahlen und Buchstaben darstellt.

Rollenbasierte Sicherheit

Die Zugriffserlaubnis auf eine Datei oder eine Ressource in der Anwendung hängt oftmals von den Anmeldeinformationen des Benutzers ab, der sich am System oder am Programm selbst angemeldet hat. An einem Beispiel lässt sich am besten erklären, wie rollenbasierte Sicherheit funktioniert. Gehen wir von einem klassischen Fertigungsbetrieb aus, der zum Beispiel Autowerkzeuge herstellt. Dieses Unternehmen setzt eine Anwendung ein, um Rohstoffe und Materialien für die Fertigung zu beziehen, die Fertigung zu steuern und den Verkauf der Produkte zu überwachen. Während die Geschäftsleitung vollständigen Zugriff auf die Bereiche Einkauf, Fertigung, Vertrieb hat, besitzt die Abteilung Einkauf nur Zugriffsrechte für den Bereich Einkauf. Die Mitarbeiter in der Herstellung können nur Informationen über den Fertigungsprozess abrufen und der Vertrieb kann nur auf explizit für ihn bestimmte Bereiche zugreifen. Die Anwendung ist somit rollenbasiert und liefert nur die Informationen, für die der einzelne Benutzer oder die Benutzergruppe Rechte hat.

Für die Autorisierung des Benutzers werden Informationen über den sogenannten `Principal` bereitgestellt. Der `Principal` wird aus einer zugehörigen

Identität erstellt und kann auf einem Windows-Benutzerkonto oder einer von Ihnen angelegten generischen Identität basieren. Ihre Anwendung kann nun eine Autorisierung des Benutzers anhand des Windows-Kontos oder anhand der Identität durchführen.

Info

Das Principal spiegelt die Rolle und die Identität des Benutzers wider.

Code Access Security

Um zu verhindern, dass schädlicher Code oder Code von unbekannten Quellen ausgeführt und somit Ihr Programm oder den Computer, auf dem das Programm ausgeführt wird, beschädigt, gibt es im .NET Framework einen Sicherheitsmechanismus mit dem Namen Code Access Security oder kurz CAS. Beim Laden einer Assembly wird diese mit einer Liste von Zugriffsrechten verglichen. Falls eine Methode der Assembly speziellen Zugriff auf eine Ressource benötigt, wird diese Liste abgerufen. Nun wird eine Prüfungsroutine initialisiert, die kontrolliert, ob jeder Aufruf in der Assembly auch die entsprechenden Rechte hat. Ist dieses bei einem Aufruf nicht der Fall, wird eine Fehlermeldung ausgegeben und die gewünschte Operation wird nicht ausgeführt. Von dem allen bekommt der Benutzer, bis auf die Fehlermeldung, nichts mit. Diese Prozedur wird im Hintergrund des .NET Framework durch die Security Engine ausgeführt.

Isolated Storage

Isolate Storage oder isolierte Speicherung ist ein Sicherheitsmechanismus, der sich um die Speicherung von Daten kümmert. Die Besonderheit hierbei ist, dass die Daten isoliert von anderen Anwendungen gespeichert werden. Ebenfalls sind die speziellen Eigenschaften der unterschiedlichen Dateisysteme, wie zum Beispiel Name des Pfads, vorhandene Laufwerke, unabhängig gehalten. Das ermöglicht es Ihnen, Ihren Anwendungen zwar grundsätzlich Speicherzugriff zu gewähren, ohne ihnen jedoch vollen Zugriff auf Ihr Laufwerk zu geben – was sich besonders bei Anwendungen empfiehlt, die mit dem Internet kommunizieren und Daten austauschen.

1.4 Neuerungen in den .NET Framework-Versionen ab 3.0

1.4.1 .NET Framework 3.0

Das .NET Framework ist eine reine Erweiterung des Framework 2.0. Die meisten Basisklassen kommen dabei aus dem .NET Framework 2.0, es wurden lediglich neue Klassen zum bestehenden Framework hinzugefügt. Die Runtime, in der Framework 3.0-Applikationen ausgeführt werden, ist auch weiterhin die identische Runtime, mit der .NET Framework 2.0-Applikationen ausgeführt wurden, nämlich die Runtime-Version 2.0.

Die wichtigsten Neuerungen im Framework 3.0 waren dabei:

- Windows Presentation Foundation (WPF)
  - Framework zur Erstellung von grafischen Oberflächen auf Basis einer Seitenbeschreibungssprache (XAML) ähnlich zu Webapplikationen
- Windows Communication Foundation (WCF)
  - Framework zur Erstellung von verteilten Applikationen auf Basis unterschiedlichster Protokolle
- Windows Workflow Foundation (WF)
  - Framework zur Erstellung von workflowbasierten Applikationen
- Windows Card Spaces
  - Neues Authentifizierungssystem, das bislang keine größere Bedeutung auf dem Markt gefunden hat

1.4.2 .NET Framework 3.5

Ebenso wie 3.0 ist auch die Version 3.5 nur ein Aufsatz auf das Framework 2.0 und natürlich auch 3.0. Kurz nach der Einführung von 3.5 erschien das Service Pack 1 für 3.5, das wesentliche weitere Neuerungen bereithielt.

Hier bekommen Sie einen kurzen Überblick über die Erweiterungen des Framework 3.5:

- Wesentliche Erweiterungen an den Programmiersprachen

  Visual Basic 9 oder C# 3.0 wurden mit etlichen neuen Sprachfeatures versehen. Eine ganze Reihe von neuen Schlüsselwörtern ist zu den Sprachen hinzugekommen, die wir später im Einzelnen noch betrachten werden.
- Anonyme Typen

  Mit dem .NET Framework 3.5 ist es möglich, Typen ohne eigene Klassendefinitionen zu definieren.
- Einführung von LINQ

  Mit LINQ wird eine neue einheitliche Datenzugriffsmöglichkeit für unterschiedliche Arten von Daten (Objektmodell, relationale Datenbanken, XML) eingeführt. Dabei bietet LINQ Möglichkeiten, die Brücke zwischen relationalen Datenbankmodellen und der objektorientierten Welt zu schlagen. LINQ gliedert sich in vier Bereiche:
  - LINQ to Objects
  - LINQ to ADO.NET
  - LINQ to XML
  - LINQ to Entities (SP1)

    Mit dem oben bereits angesprochenen SP1 wurde das sogenannte Entity Framework eingeführt, mit dem es nun auch LINQ to Entities als Nachfolger von LINQ to ADO.NET gibt.

- Integration von Ajax in ASP.NET

  Mittels Ajax (Asynchrounus JavaScript And XML) besteht die Möglichkeit, nur noch Teile einer Website zu aktualisieren und somit Performancegewinne und eine verbesserte Usability zu erreichen.

1.4.3 .NET Framework 4.0

Die momentan aktuellste Framework-Version 4.0 stellt sich als eigenständiges Framework (also nicht wie die beiden Vorgänger als Aufsatz) mit einer neuen Runtime 4.0 dar.

Neben vielen kleinen Änderungen und Erweiterungen bietet das .NET Framework 4.0 folgende Änderungen:

- Verbesserungen im Garbage Collector
- Einführung eines neuen Datentyps BigInteger
- Parallelverarbeitung

  Wesentlich einfachere Möglichkeiten bei der Entwicklung von parallel laufenden Applikationen. Einführung von PLINQ (Parallel LINQ).
- Dynamische Erweiterungen

  Mit der Einführung der Dynamic Language Runtime (DLR) gibt es für die beiden Programmiersprachen VB und C# jetzt auch dynamische Programmiermöglichkeiten, ganz nach dem Vorbild der vor allem im Web beliebten dynamischen Sprachen. Dazu wurde auch ein neuer Namespace *System.Dynamics* eingeführt, der den neuen Typ *Dynamic* enthält. Somit ist es möglich, dass der Typ eines bestimmten Objekts erst zur Laufzeit ermittelt wird.

Abbildung 1.7 zeigt die Neuerungen passend zu den Framework-Versionen.

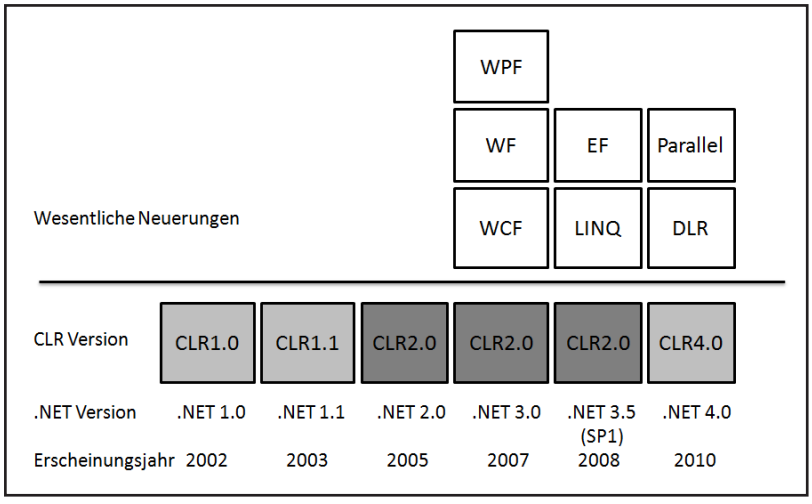


Abbildung 1.7
Neuerungen im zeitlichen Ablauf

2

Visual Studio 2010

Nach den Vorgängerversionen Visual Studio.NET 2002, 2003 und 2005 sowie Visual Studio 2008 ist Visual Studio 2010 mittlerweile die fünfte Entwicklerumgebung von Microsoft, die es ermöglicht, mit der .NET-Technologie zu programmieren. Microsoft hat einige Neuerungen und Verbesserungen in die Entwicklungsumgebung eingearbeitet, um das Programmieren so angenehm, komfortabel und effektiv wie möglich zu gestalten. Besonders ins Auge fällt dabei die völlig überarbeitete Benutzeroberfläche, die mit WPF entwickelt wurde.

2.1 Unterschiedliche Visual Studio-Versionen

Visual Studio 2010 wird in vier unterschiedlichen Versionen ausgeliefert: Visual Studio Express, Professional, Premium und Ultimate.

Als größter Unterschied zur 2008er-Produktreihe ist zu verzeichnen, dass die Teameditionen nun in der Premium- und Ultimate-Version integriert sind.

2.1.1 Express-Version

Die Express-Reihe richtet sich vor allem an Schüler, Studenten und Hobbyentwickler, welche die Programmiersprache erlernen und schnell dynamische Windows-Anwendungen und Webseiten erstellen möchten. Mit der Visual Basic 2010 Express Edition können vor allem Einsteiger mit vielen Beispielen in Visual Basic einsteigen. Sie unterstützt den gesamten Sprachumfang von Visual Basic, es handelt sich also um eine vollständige Version. Andere .NET-Programmiersprachen können damit nicht programmiert werden, diese sind aber als eigenständige Express-Versionen erhältlich. In den Express-Versionen sind jedoch viele Assistenten nicht implementiert und es besteht auch keine Möglichkeit, Add-ins zu laden.

Die Visual Basic 2010 Express-Version liegt diesem Buch bei.

2.1.2 Professional-Version

Visual Studio 2010 Professional richtet sich an fortgeschrittene Programmierer. Neben den Funktionen und der Einfachheit der Express-Reihe bietet Visual Studio 2010 Professional zusätzliche leistungsstarke Entwicklungswerkzeuge, die für die Verwendung von Client-Server-Anwendungen oder Webdiensten erforderlich sind. Mit der Professional-Version können neben WinForms-, WPF-Anwendungen und Webanwendungen auch Anwendungen für mobile Endgeräte, wie zum Beispiel Smartphones und Personal Digital Assistants (PDA), entwickelt werden.

2.1.3 Premium-Version

Mit der Visual Studio 2010 Premium-Version spricht Microsoft vor allem professionelle Entwickler an, die mit Visual Studio business-spezifische Anwendungen programmieren und bereitstellen möchten. Die Visual Studio 2010 Premium-Version vereint alle Leistungsmerkmale der Professional-Version und bietet weitere Tools wie Code Coverage, Coded UI Tests, statische Codeanalyse und Profiling.

2.1.4 Ultimate-Version

Mit Visual Studio 2010 Ultimate verfügen Sie über alle Tools, die Microsoft zu bieten hat. Zusätzlich zu den Tools der Premium-Version stehen noch weitere Werkzeuge zum Load- und Performancetest von Webanwendungen, ein historischer Debugger (IntelliTrace) sowie ein komplettes Test Management Tool zur Verfügung.

Welche Version sollte wer einsetzen? Anfänger, die mit Visual Basic 10 beginnen möchten, sollten sich die Visual Basic 2010 Express-Version zulegen. Diese bietet einen schnellen, und vor allem kostenlosen, Einstieg in die Windows Forms-Entwicklung.

Wer sich nicht auf Visual Basic festlegen oder auch Webanwendungen professionell entwickeln möchte, sollte sich die Professional-Version zulegen.

Die Premium- und Ultimate-Version ist nur dann empfehlenswert, wenn Sie in einem großen Team Software entwickeln und eine effiziente Möglichkeit suchen, den gesamten Projektablauf von der Architektur bis zum Test und zur späteren Veröffentlichung zu steuern.

2.2 Features von Visual Studio 2010

Visual Studio 2010 bringt eine Reihe von Funktionen, um den Programmierern die Arbeit so angenehm wie möglich zu machen. In diesem Abschnitt erhalten Sie einen allgemeinen Überblick über die wichtigsten davon, im nachfolgenden Kapitel wird dann noch speziell auf die Neuerungen von Visual Studio 2010 eingegangen.

Tipp

Sie können sofort nach der Installation festlegen, zu welchem Zweck Sie Visual Studio 2010 am häufigsten verwenden werden. Sie können Ihr eigenes Profil anlegen und zum Beispiel angeben, dass Sie Visual Studio 2010 hauptsächlich für die Programmierung in Visual Basic anwenden werden. Das erspart Ihnen das häufige Ändern der Einstellungen, die Sie eigentlich bei jedem Anlegen eines neuen Projekts manuell durchführen müssten.

2.2.1 Features des Editors

Microsoft arbeitet stets eng mit Programmierern zusammen, um Visual Studio zu verbessern. Viele Wünsche der Programmiergemeinde flossen auch wieder in Visual Studio 2010 ein. Hier bekommen Sie einen Überblick über die wichtigsten Features im Code-Editor von Visual Studio 2010.

IntelliSense wurde bereits bei früheren Visual Studio-Versionen bereitgestellt und erfreut sich seitdem bei vielen Programmierern großer Beliebtheit. Mit IntelliSense können Sie beispielsweise Member eines Objekts auflisten, Parameter-Info bereitgestellt bekommen, Klammern automatisch zuordnen, eine QuickInfo angezeigt bekommen oder auch ein Wort ergänzen. Um Sprachelemente zu finden oder Member eines Objekts ausfindig zu machen, müssen Sie nicht den Editor verlassen und sich beim Objekt selbst auf die Suche machen, Sie können diese Arbeit IntelliSense fast vollständig überlassen.

Bei älteren IntelliSense-Versionen war es immer noch nötig, ein Wort bzw. den Objektnamen anzugeben, nun müssen Sie nur ein paar Buchstaben eingeben und Visual Studio zeigt Ihnen sofort alle möglichen Wörter an, die zutreffen könnten.

IntelliSense ist auch für JavaScript und CSS (Cascading Style Sheets) verfügbar.

Seit der 2008er-Version ist es möglich, die eingeblendete IntelliSense-Liste durch Drücken der STRG-Taste halbtransparent darzustellen, um während der Auswahl aus der Liste den darunterliegenden Code weiterhin lesen zu können. Das stellt sich dann so dar, wie Sie es in Abbildung 2.1 sehen. Durch Loslassen der STRG-Taste wird die Liste wieder normal (und somit überdeckend) angezeigt.

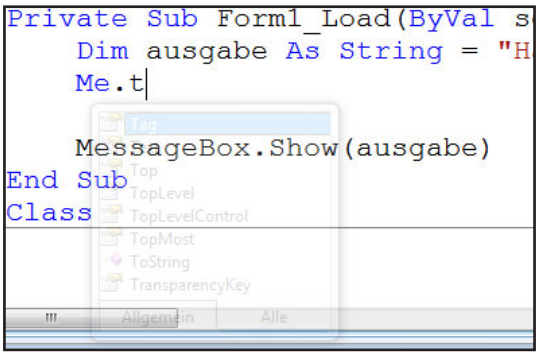


Abbildung 2.1
Halbtransparente IntelliSense-Liste

2.2.2 Multitargeting

Unter Multitargeting versteht man, dass Sie mit Visual Studio 2010, wie auch bereits in der Vorgängerversion, Projekte, die auf unterschiedlichen Framework-Versionen basieren, bearbeiten können. Davor war die Zuordnung des Visual Studio zur unterstützten Framework-Version unabdingbar.

Tabelle 2.1 gibt einen Überblick, welche Framework-Versionen von welchem Visual Studio unterstützt werden.

Tabelle 2.1 Zuordnung Framework-Version zu Visual Studio-Version

Visual Studio-Version	Framework-Version
2002	1.0
2003	1.1
2005	2.0
2008	2.0 oder 3.0 oder 3.5
2010	2.0 oder 3.0 oder 3.5 oder 4.0

Wenn Sie ein neues Projekt anlegen, können Sie durch Auswahl aus einer ComboBox die gewünschte Framework-Version auswählen (siehe Abbildung 2.2).

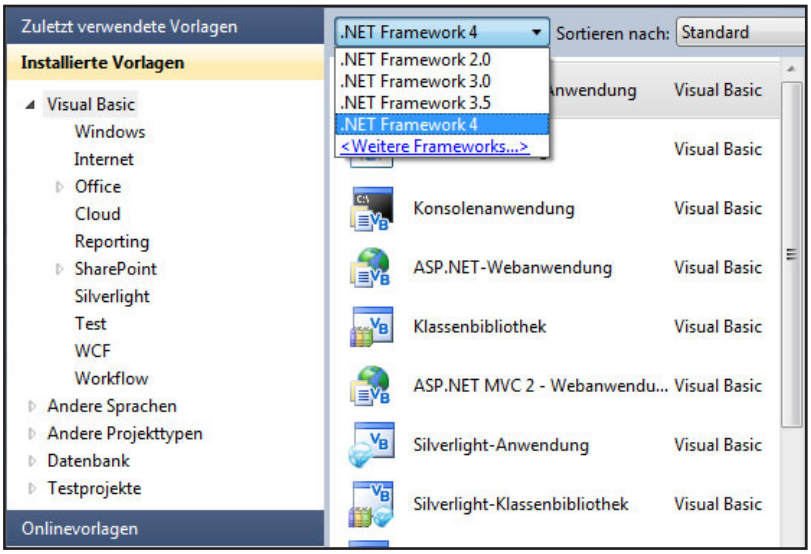


Abbildung 2.2 Auswahl der gewünschten Framework-Version

Je nachdem, welche Framework-Version Sie dabei auswählen, werden Ihnen unterschiedliche Projektvorlagen angeboten (vergleichen Sie dazu Abbildung 2.2 und Abbildung 2.3). Die IntelliSense-Liste bietet Ihnen auch nur die Funktionalität an, die von der entsprechenden Version unterstützt wird. Sie können auch keine Verweise auf Bibliotheken setzen, die nicht zu der entsprechenden Framework-Version passen.

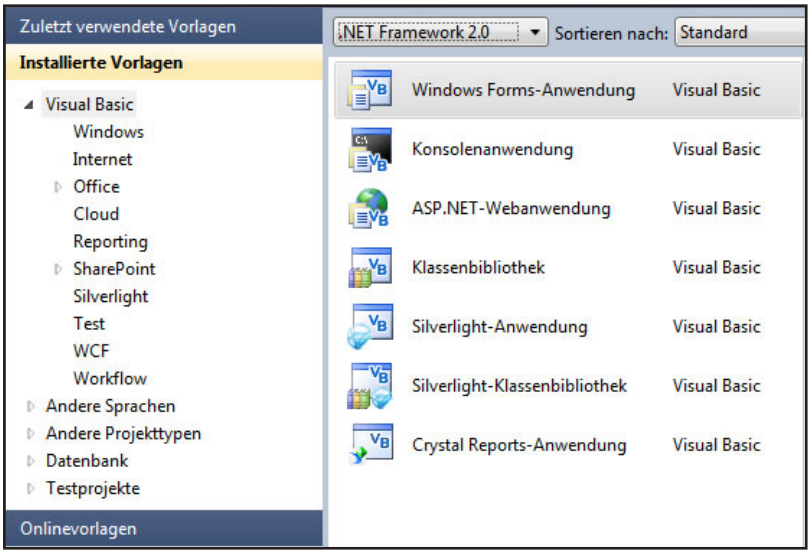


Abbildung 2.3
Projektvorlagen bei der Auswahl des Framework 2.0

Ein nachträgliches Ändern der Framework-Version ist unter den Projekteigenschaften im Register KOMPILIEREN möglich. Durch Auswählen des Buttons ERWEITERTE KOMPILIERUNGSOPTIONEN... kommen Sie zu dem in Abbildung 2.4 dargestellten Dialog, in dem Sie das Zielframework neu einstellen können.

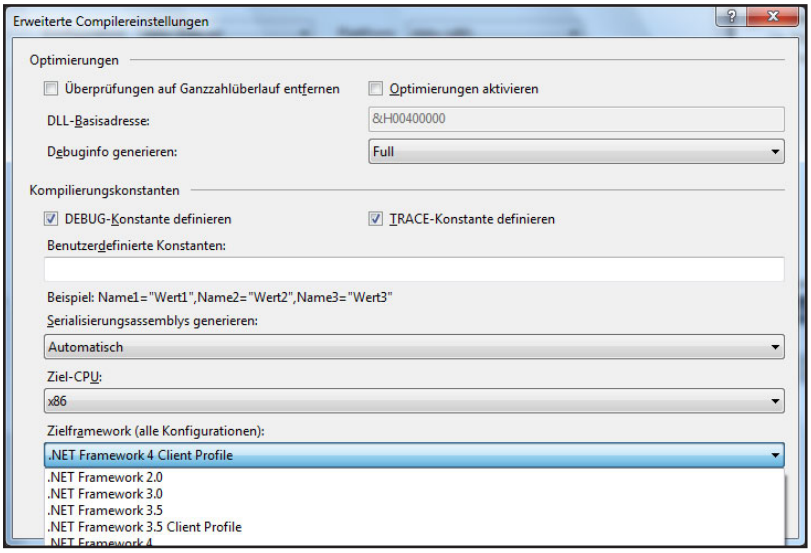


Abbildung 2.4
Erweiterte Compiler-Einstellungen zum Ändern der Framework-Version

Eine Migration nach Visual Studio 2010 bedeutet also nicht zwingend, dass Sie oder Ihre Kunden auch die Framework-Version migrieren müssen.

Achtung

2.2.3 Community und Hilfe

Das Wort Community hat bei Microsoft einen sehr hohen Stellenwert. Microsoft unterstützt die freiwilligen Entwickler in der Community in verschiedenen Programmen. Die .NET-Technologie bietet unzählige Funktionen, Klassen, Steuerelemente, so dass eine Person alleine gar nicht alles im Überblick behalten kann. Für die Lösung eigener Anwendungsprobleme ist es oft unumgänglich, im Internet zu recherchieren. Das dauert mitunter sehr lange, da man sich erst durch unzählige Suchmaschinen und Foren wühlen muss, um an den gewünschten Inhalt zu kommen.

Mit der Community Integration bietet Microsoft, neben der üblichen MSDN-Hilfe, eine Funktionalität, die auf Ihre Anfrage hin live alle bei Microsoft gemeldeten Ressourcen bei Content-Partnern durchsucht. Diese Content-Partner stellen ihre Lösungen kostenfrei ins Internet, Microsoft verlinkt diese in der Hilfe und leitet sie dann auf die entsprechende Seite um. Wie Ihnen vielleicht bereits aufgefallen ist, werden auf der Startseite von Visual Studio 2010 aktuelle Ressourcen angezeigt. Diese stammen ebenfalls aus dem Ressort der Content-Partner. Bei der Suche können Sie, neben Ihrem Suchbegriff, die gewünschte Programmiersprache und die Sprache des Inhalts anzeigen. Community und Hilfe erreichen Sie wie gewohnt über die Taste F1 oder über den Eintrag Hilfe im Visual Studio 2010-Menü.

Info

Als größte Webseite für andere .NET-Themen erweist sich *Codezone.de*. CodeZone ist eine Initiative von Microsoft Deutschland mit dem Ziel, die Sichtbarkeit und Auffindbarkeit der zahlreichen deutschsprachigen Entwicklerangebote rund um Microsoft-Technologien zu verbessern und die Transparenz dieser Ressourcen für die Developer-Community zu erhöhen. *Codezone.de* erschloss bereits zum Start im Frühjahr 2004 mehrere tausend Fundstellen zu entwicklerspezifischen Inhalten wie Artikeln, Sourcen, FAQ-Sammlungen, Büchern, Fachautoren, User Groups, Organisationen, Veranstaltungsterminen oder Online-Events und verlinkt direkt zu den Zieladressen. Mittlerweile erfolgt eine automatische Weiterleitung auf *http://social.msdn.microsoft.com/forums/de-de/categories*. *Codezone.de* ist jedoch ein bisschen einfacher in die Adresszeile des Browsers einzutippen.

2.2.4 Assistenten in Visual Studio 2010

Visual Studio 2010 bietet verschiedene Assistenten, die dem Programmierer mühselige Arbeit ersparen. Hier genannt seien vor allem der Visual Basic 6-Code-Aktualisierungsassistent sowie der Import- und Exportassistent für Einstellungen.

Code-Aktualisierungsassistent

Der Code-Aktualisierungsassistent dient dazu, alte Visual Basic 6-Projekte nach Visual Basic 10 zu konvertieren. Der Programmierer muss sein altes Projekt nicht komplett neu in Visual Basic 10 umsetzen, sondern kann einen

großen Teil migrieren und den Code dann in der neuen Umgebung gegebenenfalls weiterentwickeln. Der Code-Aktualisierungsassistent war bereits in früheren .NET-Versionen von Visual Basic enthalten, erreichte jedoch nicht immer die Zufriedenheit des Programmierers, da einige Codestücke fehlerhaft oder gar nicht konvertiert werden konnten. Seit Visual Studio 2008, das eine Neuauflage des Assistenten mit sich brachte, will Microsoft für Verbesserung sorgen. Man darf gespannt sein, was nun der Code-Aktualisierungsassistent von Visual Studio 2010 zu leisten imstande ist – Erfahrungsberichte haben bei Drucklegung dieses Buchs verständlicherweise noch nicht vorgelegen.

Import- und Exportassistent für Einstellungen (Wizard)

Falls Sie Visual Studio 20010 auf verschiedenen Maschinen benutzen bzw. beabsichtigen, Visual Studio oder Ihr Betriebssystem neu zu installieren, bietet der Import- und Exportassistent für Einstellungen die Möglichkeit, Ihre Visual Studio 2010-Einstellungen zu speichern. Sie gelangen zu diesem Assistenten über den Menüeintrag Extras im Hauptfenster von Visual Studio. Die gespeicherten Informationen werden in einer Datei mit der Endung *.vssettings* in einem beliebigen Ordner Ihrer Wahl abgelegt. Der Import erfolgt ebenfalls über diesen Assistenten. Neben der Import- und Exportfunktion bietet der Assistent die Möglichkeit, alle Einstellungen zurückzusetzen und Visual Studio von vorne neu zu konfigurieren.

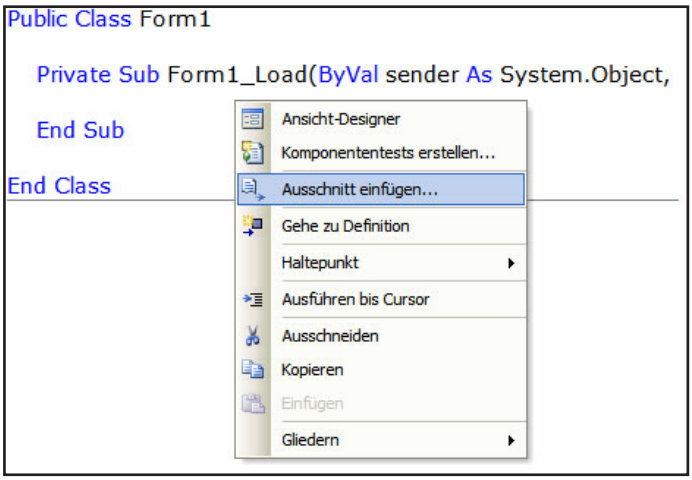
2.2.5 XML-Editor

Mit dem integrierten XML-Editor können Sie nun auch Funktionen wie Programmieren mit farblicher Hervorhebung der Schlüsselwörter oder IntelliSense verwenden, um XML-Dateien unkompliziert zu erstellen oder zu bearbeiten. Der Editor unterstützt die Prüfung der Syntax, die Validation des XML-Formats und die Möglichkeit, ein XML-Schema zu erstellen.

2.2.6 Codeausschnitte

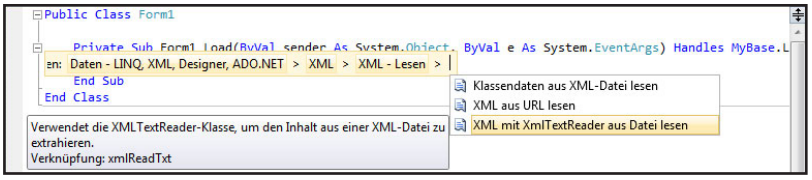
Die Code Snippets-Funktion (im Deutschen auch Ausschnitte genannt) erlaubt es Ihnen, oft verwendete Befehle und Anweisungen schnell in Ihre Anwendung einzubringen, ohne selbst eine Zeile Code schreiben zu müssen. Beim Drücken der rechten Maustaste erhalten Sie im Kontextmenü die Option Ausschnitt einfügen... Wenn Sie diese auswählen, bekommen Sie eine Liste mit verschiedenen Bereichen, die Sie per Doppelklick oder durch Drücken der Eingabetaste ansteuern können.

Abbildung 2.5
Codeausschnitt einfügen



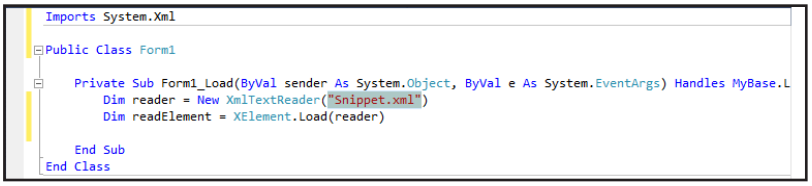
Wenn Sie beispielsweise den Inhalt aus einer XML-Datei lesen möchten, mussten Sie vor Visual Studio 2005 noch eine eigene Prozedur schreiben. Mit Code Snippets können Sie nun unter dem Punkt Daten – LINQ, XML, Designer, ADO.NET – XML – XML – Lesen – XML mit XmlTexdtReader Aus Datei Lesen wählen und der Code wird an einer beliebigen Stelle eingefügt (siehe Abbildung 2.6 und Abbildung 2.7).

Abbildung 2.6
XML-Codeausschnitt einfügen



Es wird dann direkt ein Code erzeugt, den Sie schnell anpassen können.

Abbildung 2.7
Eingefügter XML-Code



Damit der Code auch für Ihre Anwendung läuft, müssen Sie noch die unterlegten Werte ersetzen. Mit der Tabulatortaste können Sie durch die einzelnen Werte springen. Einige Codeausschnitte erhalten Verweise zur Microsoft Developer Network-Bibliothek, so dass Sie weitere Hilfe zur Verwendung mit dem Code bekommen.

NEW

In Visual Studio 2010 können Sie auch durch das Eintippen bestimmter Schlüsselwörter gefolgt von einem zweimaligen Drücken der TAB-Taste Codeausschnitte anlegen.

Ein beliebter und häufig verwendeter Codeausschnitt ist dabei die Erstellung von Eigenschaften. Dies geschieht in Visual Studio durch die Eingabe von *Property* gefolgt von zwei ⇆. Das Ergebnis sehen Sie in Abbildung 2.8.

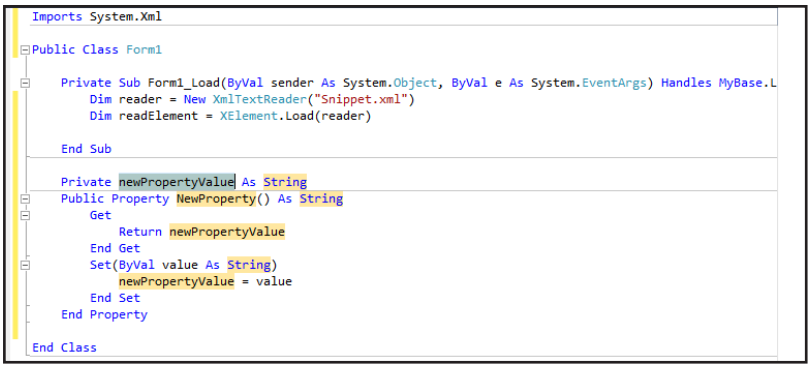


Abbildung 2.8
Anlegen einer Property mit Code Snippets

Mit der Tabulatortaste können Sie jetzt zwischen den orange hervorgehobenen Ausdrücken navigieren und sie Ihren Bedürfnissen entsprechend anpassen.

Sie können auch selbst Codeausschnitte anlegen, um diese für weitere Anwendungen zu verwenden. Die Code Snippets werden als XML-Datei in Ihrem Visual Studio 2010-Verzeichnis unter: *C:\Dokumente und Einstellungen\[Ihr Benutzername]\Eigene Dateien\Visual Studio 2010\Code Snippets\Visual Basic\My Code Snippets* gespeichert.

Um eigene Codeausschnitte anzulegen oder vorhandene zu bearbeiten, können Sie in Visual Studio unter dem Menüpunkt EXTRA den Codeausschnitt-Manager aufrufen. Hier werden alle Codeausschnitte angezeigt, die Sie in Ihren Projekten verwenden können. Die vorhandenen Codeausschnitte werden im Visual Studio-Verzeichnis unter *Vb\Snippets* gespeichert und können von hier aus in Visual Studio geladen werden.

2.2.7 Edit and Continue

Bearbeiten und Fortsetzen war bereits zu Visual Basic 6-Zeiten ein beliebtes Feature. Mit Bearbeiten und Fortsetzen oder auch Edit and Continue (E&C) konnten Sie Ihren Programmcode während des Debuggens Ihres Programms ändern. Anstatt erst das Programm und den Debugger zu beenden, können Sie die gewünschte Codestelle sofort ändern und dann mit der Programmausführung fortfahren. Dabei wird dann der angepasste Programmcode übersetzt.

Noch in den ersten beiden .NET-Versionen von Visual Studio mussten Sie Ihr Programm beenden, die gewünschte Codestelle ändern und danach das Programm wieder neu kompilieren.

Technisch funktioniert das neue Verfahren so:

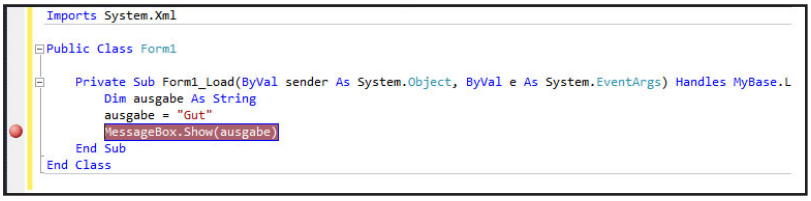
Während der Programmlaufzeit wird der Code nur im Speicher geändert. Alle bisherigen Variablen, Arrays und Objekte, wie zum Beispiel Eingaben in Textfeldern, bleiben erhalten und werden nicht von der Programmänderung berührt. Nachdem Sie die Codestelle geändert haben, wird das Programm weiter ausgeführt, jedoch mit dem von Ihnen geänderten Code.

Um die Edit and Continue-Funktion zu verdeutlichen, legen Sie in Visual Studio 2010 eine neue Windows-Anwendung an und geben Sie folgenden Code in den Form_Load-Abschnitt ein:

```
Dim ausgabe As String
ausgabe = "Gut"
MessageBox.Show(ausgabe)
```

An der Zeile `MessageBox.Show(ausgabe)` setzen Sie nun mit Rechtsklick – HALTEPUNKT – HALTEPUNKT HINZUFÜGEN – oder mit der Taste F9 einen Haltepunkt.

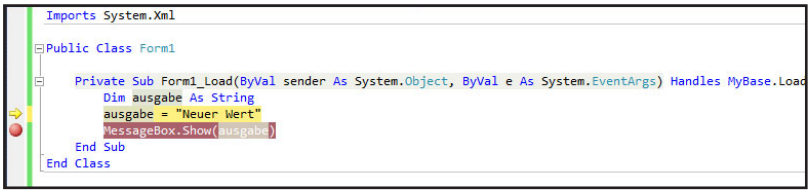
Abbildung 2.9
Edit and Continue-Haltepunkt setzen



Starten Sie dann Ihre Anwendung mit F5. Das Programm läuft bis zum Haltepunkt durch.

Dann gehen Sie zurück in den Code, ohne Ihr Programm zu beenden. Verschieben Sie den Pfeil auf die Variable `ausgabe` und ändern Sie den Wert von »Gut« in einen von Ihnen beliebig gewählten, wie ich es in Abbildung 2.10 gemacht habe.

Abbildung 2.10
Änderung des Variablenwerts zur Laufzeit



Als Ausgabe zeigt sich im Meldungsfenster der von Ihnen zur Laufzeit geänderte Wert, wie Sie in Abbildung 2.11 erkennen können.

Abbildung 2.11
Ausgabe der geänderten Variablen

Mit Edit and Continue haben Sie ein mächtiges Werkzeug, das Ihnen die Arbeit bei der Programmierung erleichtert und sie beschleunigt. Sie können fast jede Änderung am Code vornehmen, bis auf die Änderung eines Klassennamens, das Hinzufügen von Feldern zu einer Klasse, das Hinzufügen und Entfernen von Methoden und Methodenparametern.

2.2.8 Auto Correct

Seit Visual Studio 2005 steht eine Auto-Korrektur-Funktion zur Verfügung. Diese arbeitet ähnlich wie die Korrektur in Microsoft Word. Falls Sie einen Tippfehler begehen, wird dieser von Visual Studio erkannt. Ihnen wird nun ein Dialogfeld mit verschiedenen Lösungen angeboten.

Abbildung 2.12 zeigt beispielsweise, wie ein Schreibfehler bei der Deklaration einer String-Variablen erkannt und sogleich verbessert werden kann.

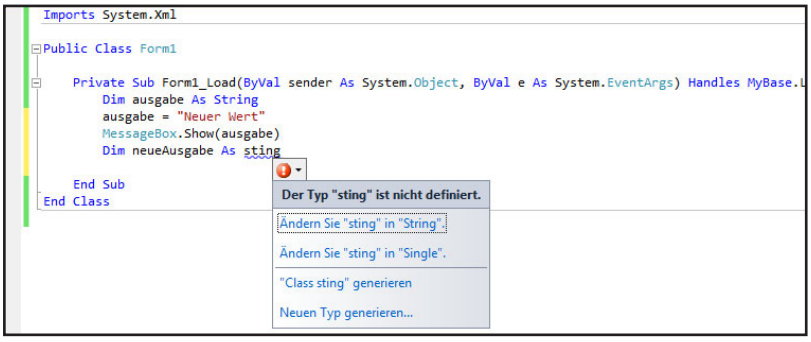


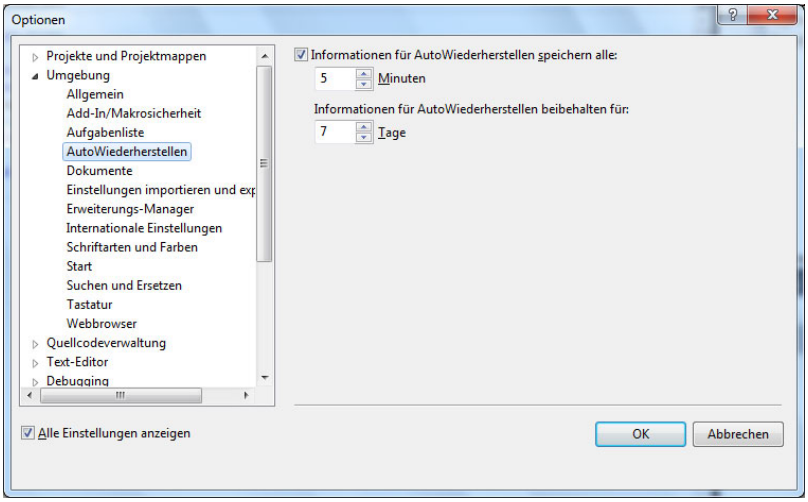
Abbildung 2.12
Autokorrektur des falsch geschriebenen Datentyps

Ebenfalls nützlich ist die Auto-Correct-Funktion bei der Verwendung von Klassen. Falls Sie vergessen haben, zu welchem Namespace eine Klasse gehört, können Sie einfach den Namen der Klasse in den Editor schreiben. Die Auto-Correct-Funktion stellt Ihnen dann verschiedene Namespaces zur Auswahl.

2.2.9 Wiederherstellen

Mit AutoRecover können Sie den Code, den Sie schreiben, automatisch zwischenspeichern. Falls Sie einmal unbeabsichtigt Code löschen oder auf älteren Code zurückgreifen wollen, können Sie das mit der AutoRecover-Funktion machen. Auf der Eigenschaftsseite, die Sie unter Extras – Optionen – Umgebung – Autowiederherstellen finden, können Sie festlegen, wie oft der Code automatisch gespeichert und wie lang dieser Code aufbewahrt werden soll, bevor er wieder gelöscht wird.

Abbildung 2.13
Einstellungen für Autowiederherstellen



Wenn Sie mit Microsoft Office arbeiten, werden Sie sicherlich bereits mit der Funktion der Dokumentwiederherstellung vertraut sein. Visual Studio 2010 bietet die Möglichkeit, durch einen Absturz nicht gesicherte Projekte einfach wiederherzustellen. In diesem Fall öffnet sich eine Meldung beim nächsten korrekten Start von Visual Studio, die Ihnen diese Option zur Verfügung stellt.

2.2.10 Umbenennen

Oftmals beginnen Sie ein Projekt und verwenden für Objekte standardisierte Namen. Später werden Sie eventuell feststellen, dass diese Namen generell nicht passend waren oder eben nicht in das Konzept passen. Visual Studio bietet hierzu eine Funktion im Kontextmenü der IDE mit dem Namen *Umbenennen*. Mit dieser Funktion können Sie alle Objekte mit denselben Namen umbenennen. Auf diese Weise vermeiden Sie im Nachhinein Syntaxfehler und die aufwändige Suche nach falsch benannten Objekten. Die genannte Funktion ist ein Teil der Refactoring-Funktionalität innerhalb von Visual Studio.

2.2.11 Debugging

Das Debuggen gilt immer noch als meist lästige Arbeit, da es sehr viel Zeit kostet, Fehler zu finden, die man bei der Programmierung nicht bedacht oder übersehen hat. Deshalb ist es wichtig, mit einer Entwicklungsumgebung zu arbeiten, die das Debuggen und das Finden von Fehlern so einfach wie möglich gestaltet.

Schon zu Visual Basic 6-Zeiten und bei den vorigen Visual Basic.NET-Versionen brauchten Sie während des Debuggens nur mit dem Mauszeiger über eine Variable zu fahren, um deren aktuellen Wert zu ermitteln. Da die Kontrolle gewöhnlicher Variablen alleine für die meisten Programmierer jedoch nicht ausreichte und ein wirklich effizientes Debuggen so nicht unbedingt möglich war, gibt es seit Visual Studio 2005 nun auch eine Unterstützung für komplexe Typen.

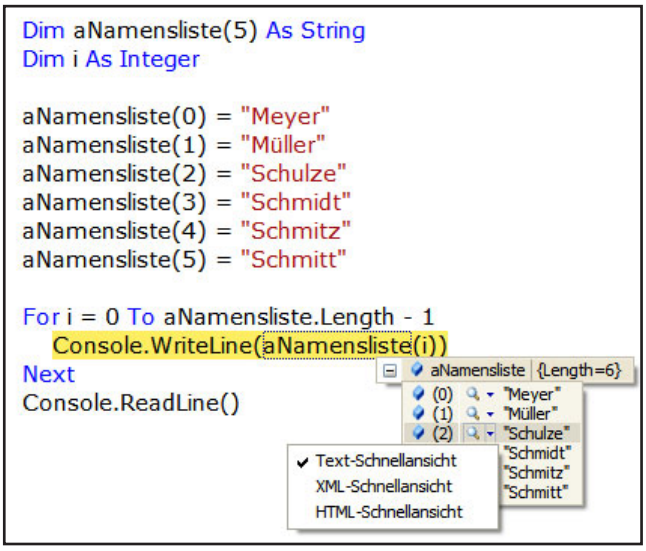


Abbildung 2.14
Komplexe Datenstruktur im Debugger

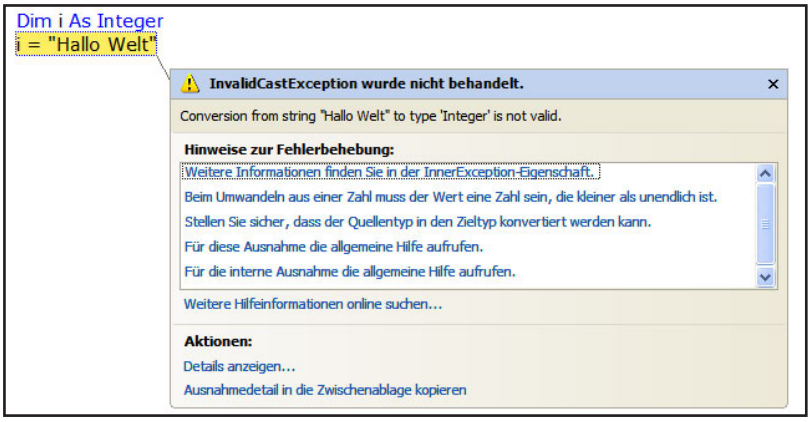
Sie können während des Debuggens, dank Edit and Continue, beispielsweise die Werte eines Arrays ändern, um so die Ausgabe zu beeinflussen. Um die Variable während der Ausführung zu beobachten, können Sie diese zum sogenannten *Watch*-Fenster hinzufügen. Dieses wird direkt in Visual Studio angezeigt und Sie behalten den Überblick, falls sich der Wert der Variablen oder der Wert eines anderen Objekts ändern sollte.

2.2.12 Fehler-Assistent

Der Fehler-Assistent war eine neue Entwicklung, die mit Visual Studio 2005 eingeführt wurde. Er erscheint, sobald beim Debuggen ein Fehler auftritt. Im Titel wird der Fehlertyp angezeigt, darunter wird eine kurze Beschreibung des Fehlers dargestellt. Falls weitere Informationen zum Fehler verfügbar sind,

wird eine Liste mit diesen Themen angezeigt. Dahinter verbirgt sich dann ein Link, der auf die MSDN-Hilfe verweist. Ein Beispiel sehen Sie in Abbildung 2.15.

Abbildung 2.15
Fehler-Assistent im Einsatz



2.2.13 ClickOnce-Installation

Sie können mit der ClickOnce-Installation festlegen, von welcher Stelle aus Ihre Anwendung installiert werden soll. Hierbei stehen Ihnen verschiedene Möglichkeiten zur Auswahl. Zum einen können Sie Ihre Anwendung von einem Datenträger, von einem Netzlaufwerk oder von einem Web-Speicherort installieren. Bei der Webinstallation wird über eine Webseite die erforderliche Setup-Datei heruntergeladen und zur Installation gestartet. Bei diesem Schritt werden dann alle erforderlichen Einträge, wie zum Beispiel die nötigen Verknüpfungen im Startmenü, erstellt. Es ist ebenfalls möglich, ClickOnce-Anwendungen so zu erstellen, dass sie nicht auf dem Rechner des Anwenders installiert, sondern auf dem Server selbst ausgeführt werden. Selbstverständlich können Sie, wie in den vorigen .NET-Versionen, auch noch normale Setup-Dateien mit den gewöhnlichen Installationsroutinen erstellen. Der große Vorteil von ClickOnce-Installationen besteht darin, dass sich die Applikationen automatisch aktualisieren können.

Mehr zu ClickOnce lesen Sie in Kapitel 11, Deployment.

2.2.14 Temporäre Projekte

In den ersten beiden Visual Studio-Versionen war es notwendig, sofort beim Anlegen von Projekten einen Speicherort auszuwählen. Das war teilweise überflüssig, wenn Sie nur eine Kleinigkeit testen wollten. Ab der 2005er-Version können Sie Projekte erst einmal temporär und erst später bei Aufruf der Speicheroption an einem angegebenen Ort speichern. Sie erreichen diese Einstellungsmöglichkeit über Extras, darin den Menüpunkt Optionen, gefolgt von Projekte und Projektmappen. Wählen Sie hier das Register Allgemein und

deaktivieren Sie die Option Neue Projekte beim Erstellen speichern, wenn neue Projekte standardmäßig temporär erstellt werden sollen.

2.2.15 XML-Dokumentation

Eine saubere Dokumentation ist ein wichtiger Bestandteil einer jeden Applikation. Nicht nur, wenn mehrere Programmierer an einer Applikation entwickeln, auch für Sie selbst ist es wichtig, alle Bestandteile eines Projekts zu dokumentieren, gerade dann, wenn zwischen der Entwicklung eine längere Pause liegt. Leider kann Visual Studio Ihnen nicht die Dokumentationserstellung abnehmen. Es bietet jedoch ein Feature für Visual Basic-Entwickler, das C#-Entwickler bereits aus der ersten .NET-Version kennen. Mit dem XML-Dokumentationsschema dokumentieren Sie jede Ihrer Applikationen in ein und derselben Art. Das erleichtert es Ihnen nachher, aus der Dokumentation mit anderen Tools eine ausführliche Problem- und Reportanalyse zu erstellen. Diese XML-Kommentare unterscheiden sich, im Gegensatz zu Ihnen bekannten Kommentaren, durch ein eigenes Format.

Eine Dokumentation beginnt immer mit drei Apostrophen (`'''`). Wenn Sie in Ihrem Projekt diese drei Apostrophe eingeben, wird automatisch das Dokumentationsschema erstellt.

Nehmen wir zum Beispiel folgende Funktion:

```
Private Function FormulareZaehlen()
    Dim i As Integer
    For Each frm As Form In My.Application.OpenForms
        MessageBox.Show(frm.Text)
        i += 1
    Next
    Return i
End Function
```

Wenn Sie nun vor die Funktion drei Apostrophe einfügen, wird automatisch folgende Dokumentation ergänzt:

```
''' <summary>
'''
''' </summary>
''' <returns></returns>
''' <remarks></remarks>
```

Dieses Konstrukt enthält als erstes Element für eine kurze Zusammenfassung des Projekts den Tag *<summary>*.

<remarks> dient als Erweiterung zum *Summary*-Tag. Damit können Sie Klassenmember näher beschreiben.

Für jeden Parameter, der in der Klasse vorhanden ist, wird ein einzelner *<param>*-Tag angelegt.

<returns> ist für Rückgabewerte von Funktionen vorgesehen.

Innerhalb von *<exception>* können die Fehler, die von der Funktion zurückgegeben werden, aufgelistet und beschrieben werden.

<example> kann ein Beispiel zur Verwendung der Funktion darstellen.

In *<see>* können Sie auf andere Dokumentationselemente verweisen.

Des Weiteren können Sie Tags verwenden, die für die stilistische Ausführung verantwortlich sind. Hierzu zählen:

- <para>

  Für die strukturelle Festlegung innerhalb des Textes
- <list>

  Für eine Auflistung innerhalb der Elemente
- <c> oder auch <code>

  Für den Hinweis auf eine Codezeile bzw. mehrere Codezeilen in der Dokumentation

Folgende Dokumentation wurde mit Inhalt zur Funktion gefüllt:

```
''' <summary>
''' Durchläuft alle offenen Fenster der Applikation und
''' <list>Zählt diese</list>
''' <list>Gibt deren Namen in einer MessageBox aus</list>
''' </summary>
''' <returns>i</returns>
''' <remarks></remarks>
```

Diese Dokumentation wird bei der Kompilierung in den Metadaten der Assembly gespeichert und kann im Objektbrowser aufgerufen werden.

Sie können übrigens die XML-Dokumentation ebenfalls in einer XML-Datei exportieren. Hierzu müssen Sie doppelt auf den *My Project*-Ordner klicken und das KOMPILIEREN-Register auswählen. Überzeugen Sie sich davon, dass hier die Option XML-DOKUMENTATIONSDATEI GENERIEREN, wie in Abbildung 2.16 zu sehen, ausgewählt ist. Die XML-Datei wird nun im *bin*-Ordner des Projekts gespeichert.

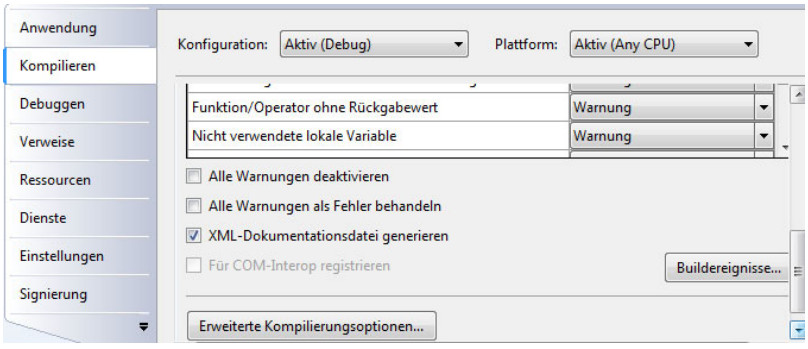


Abbildung 2.16
Option zur Festlegung der XML-Dokumentation

Tipp

Leider bietet Visual Studio 2010 selbst kein Programm an, um eine Hilfe-Dokumentation, wie man es von den meisten anderen Anwendungen gewohnt ist, zu erstellen. Hierfür gibt es aber andere Programme, die genau das für Sie tun. Hier lohnt es sich, einmal einen näheren Blick auf das Programm *Sandcastle (http://sandcastle.codeplex.com/)* zu werfen. Dieses Programm erlaubt Ihnen das Erstellen einer MS Help 2.0-Dokumentation mit Hilfe der von Visual Studio ausgegebenen XML-Tags.

2.3 Neuerungen in Visual Studio 2010

Von den vielen Kleinigkeiten, die das Leben in der neuen Visual Studio-Version angenehmer machen, wollen wir die aus unserer Sicht wichtigsten Features vorstellen.

Verbessertes IntelliSense

IntelliSense hat in den Vorgängerversionen nur funktioniert, wenn der entsprechende Begriff beginnend mit dem ersten Zeichen korrekt geschrieben wurde. Oft hat man jedoch nur einen Teil eines Klassennamens oder einer Methode im Kopf, ohne sich an die ersten Zeichen des Namens erinnern zu können. Das IntelliSense in der aktuellen Version funktioniert nun auch, wenn der entsprechende Suchbegriff nur partiell angegeben wird.

Abbildung 2.17 zeigt, dass bei der Eingabe von *Exce* (ich suche nach einer `Exception`) nicht nur Vorschläge in der Liste auftauchen, die mit der entsprechenden Zeichenfolge beginnen, sondern auch solche, die diese Zeichenfolge beinhalten.

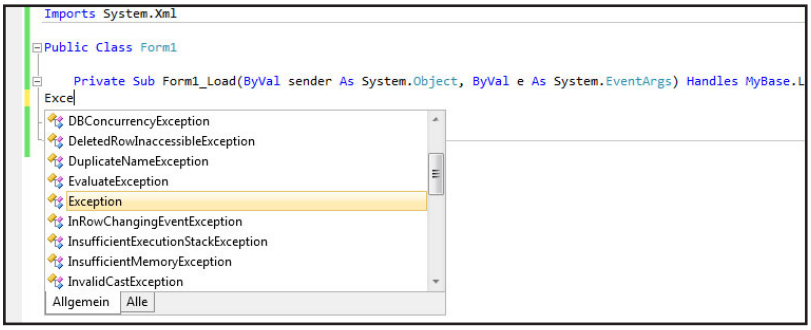


Abbildung 2.17 Verbessertes IntelliSense in Action

Zoomfunktionalität

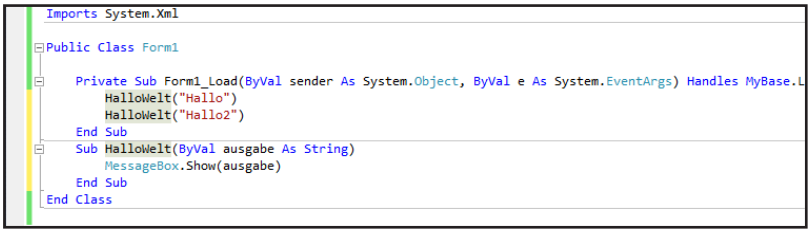
Sie können jetzt im Editor den Code zoomen und somit zum Beispiel für Präsentationen vergrößern. Drücken Sie dazu einfach die Strg-Taste und zoomen Sie mit dem Mausrad, bis die gewünschte Darstellung groß bzw. klein genug ist.

Referenzhervorhebung

Wenn Sie in einer Codedatei eine Methode markieren, werden automatisch alle anderen Vorkommen dieses Methodenaufrufs sowie – sofern vorhanden – die Definition der Methode innerhalb derselben Codedatei hervorgehoben. Dies illustriert an einem kleinen Beispiel Abbildung 2.18.

Mit Strg+⇧+Down bzw. Strg+⇧+Up können Sie zwischen den Aufrufen hin und her navigieren.

Abbildung 2.18
Referenzhervorhebung

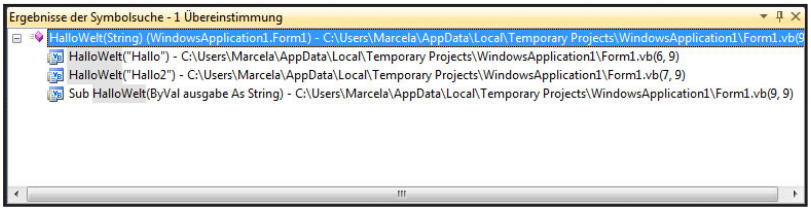


Aufrufhierarchie

Wenn Sie auf eine Methode, Eigenschaft oder einen Konstruktor mit der rechten Maustaste klicken, finden Sie im Kontextmenü den Befehl Alle Verweise suchen. Mit dieser Funktion können Sie sich alle Stellen im Programm anzeigen lassen, an denen die entsprechende Methode/Eigenschaft oder der Konstruktor aufgerufen wird.

Mit einem Doppelklick in der Trefferliste wird sofort der entsprechende Aufruf aktiviert.

Abbildung 2.19
Alle Aufrufe von HalloWelt in der Aufrufhierarchie



Boxselektion

Mittels der Boxselektion können Sie gleiche Änderungen an mehreren Zeilen Code gleichzeitig vornehmen.

Stellen Sie sich vor, Sie haben innerhalb einer Klasse mehrere Variablen als `Integer` definiert und wollen deren Datentyp nachträglich auf `Double` ändern. Anstatt diese Änderung jetzt Zeile für Zeile durchzuführen, können Sie dies mit dem neuen Feature auf einmal machen. Markieren Sie dazu den gewünschten Bereich mit ⇧+Alt und den entsprechenden Pfeiltasten und führen Sie Ihre gewünschten Änderungen durch. Dieses Vorgehen ist in Abbildung 2.20 und Abbildung 2.21 dargestellt.

```
Public Class Form1
    Private zahl1 As Integer
    Private zahl2 As Integer
    Private zahl3 As Integer
    Private zahl4 As Integer
    Private zahl5 As Integer
```

Abbildung 2.20
Auswahl der zu ändernden Stellen in der Box

```
Public Class Form1
    Private zahl1 As doubl
    Private zahl2 As doubl
    Private zahl3 As doubl
    Private zahl4 As doubl
    Private zahl5 As doubl
```

Abbildung 2.21
Durchgeführte Änderung (fast fertig) in der Box

3

Visual Basic 10 allgemein

Ein Schwerpunkt dieses Buchs liegt darin, die wichtigsten sprachlichen Merkmale von Visual Basic 10 aufzugreifen und Ihnen näherzubringen, wie man mit diesen umfangreiche Applikationen schreiben kann. Dieses Kapitel behandelt allgemeine Dinge, die für das Programmieren notwendig sind. Visual Basic wird von Kritikern oft als simple Programmiersprache bezeichnet. Doch seit dem .NET Framework ist es ohne Weiteres möglich, auch komplexe Applikationen mit Visual Basic zu entwickeln. Falls Sie noch nicht mit Visual Basic unter .NET vertraut sind oder gerade von Visual Basic 6 umsteigen, kann Ihnen dieses Kapitel die wichtigsten Grundkenntnisse in Visual Basic 10 vermitteln.

3.1 Datentypen

Variablen gehören zu den wichtigsten Bestandteilen einer jeden Programmiersprache. Variablen werden mit einem Datentyp definiert, der festlegt, welche Art von Daten in der Variablen gespeichert werden können. Die allgemeine Syntax lautet:

```
Dim VariablenName As Datentyp
```

Visual Basic 10 achtet ganz genau darauf, ob der Wert, der in die Variable geschrieben wird, auch tatsächlich zum Datentyp passt.

Folgender Code würde eine Compiler-Fehlermeldung hervorrufen, da der Datentyp *Integer* nur für ganze Zahlen gedacht ist und keinen Text speichern kann.

```
Dim Name As Integer
Name = "Max Mustermann"
```

Visual Basic 2010 kennt die in Tabelle 3.1 aufgelisteten Datentypen:

Tabelle 3.1
Datentypen im Überblick

Datentyp	Bedeutung
Boolean	Wahr-/Falsch-Wert (True/False)
Byte	Ganzzahl zwischen 0 und 255
Char	Unicode-Zeichen
Date	Datumswert vom 1.1.0001 bis zum 31.12.9999
Decimal	Eine Ganzzahl mit 128 Bit mit maximal 28 Stellen nach dem Trennzeichen
Double	Fließkommazahl mit 15 Stellen nach dem Komma
Integer	Ganzzahl mit der Größe von 4 Byte
Long	Ganzzahl mit der Größe von 8 Byte
Object	Basisklasse aller komplexen Typen
SByte	Ganzzahl zwischen -128 und 127
Short	Ganzzahl mit der Größe von 2 Byte
Single	Fließkommazahl
String	Zeichenkette mit bis zu 2 Milliarden Zeichen
UShort	Positive Ganzzahl zwischen 0 und 65535
UInt	Positive Ganzzahl mit 4 Byte
ULong	Positive Ganzzahl mit 8 Byte
BigInteger	Für Ganzzahlen, die außerhalb des Long-Bereichs liegen

Jeder Typ hat einen eigenen Namen (z.B. String) und eine bestimmte Größe. Die Größe gibt an, wie viele Bytes jedes Objekt im Speicher belegt. Jeder Datentyp in Visual Basic 10 gehört zu einem .NET-Datentyp. Was Visual Basic *Integer* nennt, ist in .NET unter *Int32* bekannt. Das ist für Sie von Bedeutung, falls Sie vorhaben, Objekte für mehrere Programmiersprachen im .NET Framework bereitzustellen. Sie können innerhalb von Visual Basic sowohl die VB-Bezeichnung angeben wie auch den zugrunde liegenden .NET-Typen.

3.1.1 Variablen

Eine Variable ist kurz gesagt ein Objekt, das einen bestimmten Wert enthält. Zum Beispiel:

```
Dim text As String = "Hallo Welt"
```

In diesem Beispiel ist `text` ein `String`-Datentyp. Bei der Initialisierung wird der Variablen der Wert `"Hallo Welt"` zugeordnet. Die eigentliche formale Definition, wie eine Variable deklariert wird, lautet:

```
Access-Modifikator Bezeichner As Datentyp (= Wert)
```

Bezeichner ist der Fachausdruck für die Namen von Elementen, die im Code vorkommen können, wie zum Beispiel Klassen, Methoden oder eben Variablen. Hier bezieht sich dieser Begriff auf den Variablennamen. Im weiter oben genannten konkreten Beispiel ist der Bezeichner `text`.

Info

`Dim` ist die Abkürzung für Dimension. Diese Abkürzung existiert beispielsweise auch in der Mathematik und in der Physik. Sie lässt sich auf die alten Tage der BASIC-Programmierung zurückführen.

Es ist nicht immer notwendig, bei der Deklaration einer Variablen einen Wert zuzuweisen. Falls Sie das nicht tun, wird Visual Basic 10 in Abhängigkeit vom Datentyp Standardwerte vergeben. Diese können Sie der Tabelle 3.2 entnehmen:

Tabelle 3.2
Initialwerte von Datentypen

Datentyp	Wert
Boolean	False
Date	01/01/0001 12:00:00 AM
Decimal	0
Object	Nothing
String	""
Alle numerischen Typen	0

Nachdem nun einige Datentypen vorgestellt wurden, will ich noch kurz darauf eingehen, wie diese Typen im Speicher repräsentiert werden.

Prinzipiell gibt es zwei unterschiedliche Arten von Datentypen: Wertetypen und Referenztypen.

Bei den Wertetypen werden die Daten direkt im Speicher abgebildet, wie es bei den bislang vorgestellten Datentypen, mit Ausnahme von Strings, der Fall ist. Hinter der Variablen steht also direkt deren Wert.

Die zweite Art, Daten zu repräsentieren, erfolgt über Referenzen auf die eigentlichen Daten. Dies ist bei komplexen Datentypen, bei denen die tatsächliche Größe variabel oder unbestimmt ist, der Fall. Zu diesen Datentypen gehören Strings, aber auch Arrays und die benutzerdefinierten Datentypen. Man spricht deshalb von Referenztypen.

3.1.2 Wertetyp- und Referenztypsemantik

Anwendungen nutzen im RAM prinzipiell zwei Bereiche, den Stack und den Heap. Nur die Variablen im Stack besitzen Namen, Objekte im Heap sind namenlos. Dort kann man nur über Adressen auf Daten zugreifen.

Wie gerade erwähnt, unterscheiden sich Variablen in zwei Gruppen: Wertetypen und Referenztypen.

Bei den Wertetypen wird der Wert der Variablen direkt an der Speicherstelle im Stack gespeichert.

Bei den Referenztypen wird an der Speicherstelle im Stack auf eine Adresse im Heap referenziert. Sie werden somit als Zeiger auf einen anderen Speicherbereich implementiert.

Doch Wertetypen und Referenztypen unterscheiden sich nicht nur in der Art der Speicherung, sondern auch in der Art und Weise, wie Daten zugewiesen werden.

Wertetypsemantik

Bei der Zuweisung eines Wertetyps wird eine Kopie der entsprechenden Variablen angefertigt. Wird die zugewiesene Variable manipuliert, hat das keine Auswirkungen auf die Originalvariable.

```
Dim zahl1 As Integer = 7
Dim zahl2 As Integer = zahl1
zahl2 = 9
```

In diesem Beispiel wird eine Variable `zahl1` vom Typ `Integer` definiert und dabei der Wert 7 zugewiesen. Im nächsten Schritt wird eine zweite Integervariable `zahl2` definiert und ihr wird der Wert von `zahl1` zugewiesen. Der Wert von `zahl1` wird aufgrund der Wertetypsemantik in die Speicherzelle `zahl2` kopiert. Anschließend wird der Wert von `zahl2` auf 9 gesetzt. Wenn beide Variablen danach ausgegeben werden, dann haben sie auch unterschiedliche Werte. Einmal den Wert 7 für `zahl1` und 9 für `zahl2`. Sie sehen, dass bei der Zuweisung `zahl2 = zahl1` eine Kopie des Werts durchgeführt wurde.

Referenztypsemantik

Bei der Zuweisung eines Referenztyps hingegen wird der Wert der Variablen nicht kopiert. Vielmehr wird die Adresse im Heap, die auf das tatsächliche Objekt zeigt, kopiert. Somit zeigen beide Variablen auf denselben Speicherbereich. Wird die zugewiesene Variable manipuliert, hat das Auswirkungen auf die Originalvariable.

```
Dim p1 As New Person()
p1.Name = "Julia"
Dim p2 As Person = p1
p2.Name = "Lennard"
```

In diesem Beispiel wird eine Variabel `p1` vom Typ `Person` (ohne der Objektorientierung vorgreifen zu wollen, sehen Sie die Definition dieses Typs in Listing 3.1) definiert und instanziert. Dann wird der Eigenschaft `Name` der Wert `"Julia"` zugewiesen. Im nächsten Schritt wird eine zweite Personenvariable `p2` definiert und ihr wird `p1` zugewiesen. Dabei wird die Adresse von `p1` aufgrund der Referenztypsemantik in die Speicherzelle `p2` kopiert. Anschließend wird die Name-Eigenschaft von `p2` auf `"Lennard"` gesetzt. Wenn von beiden Variablen danach die Name-Eigenschaft ausgegeben wird, dann haben sie beide identische Werte, nämlich `"Lennard"`. Bei der Zuweisung `p2 = p1` wurden also nicht etwa die eigentlichen Daten des Person-Objekts, sondern allein die Adresse kopiert.

```
Public Class Person
    Private _name As String
    Public Property Name() As String
        Get
            Return _name
        End Get
        Set(ByVal value As String)
            _name = value
        End Set
    End Property
End Class
```

Listing 3.1 Definition einer Klasse Person

Achtung

Obwohl es sich beim Datentyp `String` um einen Referenztyp handelt, verhält er sich wie ein Wertetyp. `String` ist somit ein Referenztyp mit Wertetypsemantik. Die Ausnahme wurde von den Framework-Entwicklern gemacht, um die Verarbeitung dieses doch wesentlichen Datentyps intuitiver zu implementieren.

3.1.3 Nullable Typen

Die vorher gerade vorgestellten Wertetypen werden bei der Definition mit einem Standardwert versehen, sofern Sie nicht direkt einen Initialwert zuweisen (siehe Tabelle 3.2).

Wertetypen besitzen somit immer einen Wert, auch wenn es eben nur der Initialwert ist. Ein Wertetyp kann niemals den Wert `Nothing` annehmen. Doch gerade bei Datenbankanwendungen ist es sinnvoll, Spalten, die den Wert `null` (nicht belegt) besitzen dürfen, nicht mit einem Initialwert zu belegen. Stellen Sie sich vor, in einer Mitarbeitertabelle gibt es für einen Mitarbeiter ein Feld *AnzahlKinder*. Wenn der Wert nicht bekannt ist, wird eben nichts eingetragen, also `null`, und somit weiß jeder, dass diese Information nicht bekannt ist. Wird jedoch der Initialwert eingetragen, würde man daraus schließen, dass der Mitarbeiter keine Kinder hat, was unter Umständen eine falsche Information darstellt.

Mit dem .NET Framework 2.0 wurden sogenannte *Nullable Types* eingeführt. Das bedeutet, dass Wertetypen jeweils ein Nullable-Äquivalent besitzen, das auch den Wert `Nothing` annehmen kann.

Einen **Nullable Integer** definieren Sie dabei wie folgt:

```
Dim Arg1 As Nullable(Of Integer)
```

NEW

Seit der Einführung der Vorgängerversion Visual Basic 9 können Sie die Definition jetzt auch wie folgt angeben:

```
Dim Arg1 as Integer?
```

3.1.4 Konstanten

Konstanten beinhalten beliebige Werte, die sich, im Gegensatz zu Variablen, über die gesamte Laufzeit des Programms nicht ändern. Der Vorteil von Konstanten ist, dass bei festen Werten eine Änderung meist durch Austauschen und Anpassung einer Zeile möglich ist. Falls Sie zum Beispiel an einem Programm arbeiten, das die Besoldung von Mitarbeitern nach einem festen Stundenlohn ausrechnet, wäre es ratsam, diesen festen Stundenlohn als Konstante festzulegen. Andernfalls müssten Sie wahrscheinlich mehrmals im Code den entsprechenden Stundenlohn als Zahl eingeben. Bei Änderung des Stundenlohns würde im zweiten Fall eine aufwändige Änderung des Codes nötig sein, während Sie bei einer Konstante nur einen Wert ändern müssten.

Beispiele für die Definition von Konstanten sehen Sie in Listing 3.2:

Listing 3.2 Festlegung von Konstanten

```
Const STUNDENLOHN As Integer = 23
Const ARBEITSSTUNDEN As Integer = 8
Const LOHNPROTAG As Integer = _
    STUNDENLOHN * ARBEITSSTUNDEN
Const WindowsPfad As String = "C:\Windows\"
```

Konstanten können, wie Variablen, direkt im Programm mit ihrem Namen angesprochen werden. Der Wert einer Konstanten kann nach der Definition nicht mehr verändert werden. Ein entsprechender Versuch würde vom Compiler mit einer Fehlermeldung beantwortet.

3.1.5 Aufzählungen (Enums)

Enums helfen Ihnen, konstante Werte einer Gruppe zuzuordnen. Nehmen wir an, Sie entwickeln eine Applikation für die Verwaltung von Mitarbeitern. Hierzu teilen Sie die Mitarbeiter in verschiedene Kategorien wie Fertigung, Einkauf, Verkauf, Marketing etc. ein. Sie können nun eine Enum als eine Art Liste anlegen.

```
Public Enum KatMitarbeiter
    Fertigung
    Einkauf
    Verkauf
    Marketing
End Enum
```

Falls von Ihnen nicht anders festgelegt, ordnet das .NET Framework nun jedem der Auflistungswerte einen numerischen Wert zu, angefangen bei 0 in aufsteigender Reihenfolge. Alternativ können Sie im Code auch direkt die Enum-Konstanten verwenden, natürlich mit IntelliSense-Unterstützung. Das hat unter anderem den Vorteil, dass Sie sich die Bedeutung der Indizes nicht merken müssen, und es erhöht auch die Lesbarkeit des Quellcodes.

Das folgende Beispiel (Listing 3.3) soll die Funktionsweise von Enums verdeutlichen:

Listing 3.3
Verwendung einer Enum

```
Public Enum KatMitarbeiter
    Fertigung
    Einkauf
    Verkauf
    Marketing
End Enum

Public Sub MitarbeiterEnum()
    Dim mitarbeiter As KatMitarbeiter
    mitarbeiter = KatMitarbeiter.Verkauf
    MessageBox.Show(mitarbeiter)
End Sub
```

Bei Ausführung des Programms wird die MessageBox den Wert 2 ausgeben, da der Name `Verkauf` in der Enum den Wert 2 enthält.

Falls Sie ermitteln wollen, welches Element sich hinter einem Wert in der Aufzählung verbirgt, lässt sich das recht simpel bewerkstelligen, indem Sie an der Enumerationsvariablen die `ToString()`-Methode aufrufen:

```
Public Sub MitarbeiterEnum()
    Dim mitarbeiter As KatMitarbeiter
    mitarbeiter = KatMitarbeiter.Verkauf
    MessageBox.Show(mitarbeiter.ToString)
End Sub
```

Solange nicht anders festgelegt, werden die Elemente als Integer angelegt. Falls gefordert, können Sie auch einen anderen Datentyp für die Aufzählungselemente festlegen. Eine Definition als Byte, Short oder Long reicht vollkommen aus.

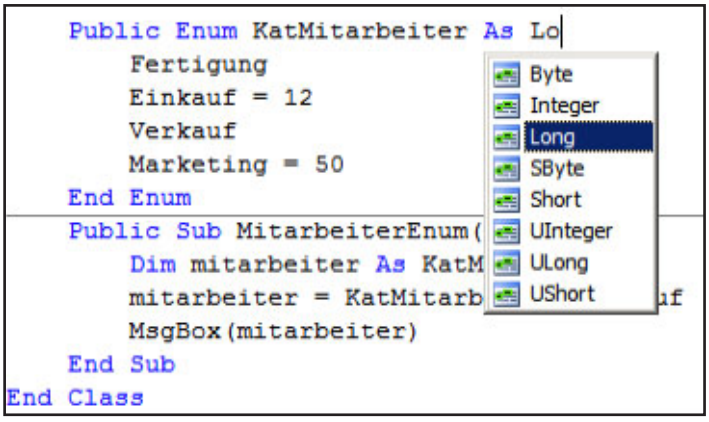


Abbildung 3.1
Definieren des Datentyps in einer Enum

Falls Sie, aus welchen Gründen auch immer, nicht mit der Zählung der Enums zufrieden sind, können Sie auch selbst festlegen, welche Nummer einem Enumerationswert zugeordnet werden soll.

```
Public Enum KatMitarbeiter
    Fertigung
    Einkauf = 12
    Verkauf
    Marketing = 50
End Enum
```

In diesem Beispiel würde die Ausgabe der Werte wie in Tabelle 3.3 aufgelistet lauten:

Tabelle 3.3 Enum-Übersicht

Enum	Wert
Fertigung	0
Einkauf	12
Verkauf	13
Marketing	50

Was aber nun, wenn einige Mitarbeiter verschiedene Aufgaben im Betrieb haben, beispielsweise im Einkauf und Verkauf gleichzeitig beschäftigt sind? Für diesen Fall liefert Visual Basic natürlich auch eine Lösung, sogenannte `Flags`-Enums oder zu Deutsch Flags-Aufzählungen.

Sie sollten hierbei jedem Eintrag einen Wert zuweisen, der sich bitweise kombinieren lässt. Ebenfalls sollten Sie einen neuen Eintrag mit dem Namen None anlegen, der den Wert 0 enthält.

```
<Flags()> Public Enum KatMitarbeiter
    Fertigung = 1
    Einkauf = 2
    Verkauf = 16
    Marketing = 32
    None = 0
End Enum
```

Nun kann eine Enumerationsvariable gleichzeitig mehrere Enum-Werte halten. Diese können Sie zum Beispiel mit dem bitweisen Or-Operator abfragen. Der Code in Listing 3.4 soll das illustrieren:

Listing 3.4 Abfrage von Enum-Werten mit Or

```
Public Sub MitarbeiterEnum()
    Dim mitarbeiter As KatMitarbeiter
    mitarbeiter = _
KatMitarbeiter.Verkauf Or KatMitarbeiter.Marketing
    If mitarbeiter = KatMitarbeiter.Verkauf Or
        KatMitarbeiter.Marketing Then

        MessageBox.Show(mitarbeiter.ToString)
        'Ausgabe: Verkauf, Marketing
    End If
End Sub
```

3.1.6 Konvertierung in verschiedene Datentypen

Visual Basic 10 bietet die Möglichkeit, Objekte in verschiedene Datentypen zu konvertieren. Nehmen wir an, Sie haben eine *String*-Variable mit dem Inhalt `"24"` und möchten damit eine Berechnung durchführen. Wie Sie sicherlich wissen, ist es zwar möglich, mit Zeichenketten zu rechnen (diese werden vor Verwendung in den Typ `Integer` konvertiert), jedoch würde die `Option Strict On`-Eigenschaft das nicht erlauben (zu `Option Strict On` erfahren Sie im späteren Teil dieses Kapitels mehr). Daher wäre es ratsam, den Wert der Variablen vor der Berechnung nach `Integer` zu konvertieren. Das funktioniert so, wie Sie es in Listing 3.5 sehen:

Listing 3.5
Konvertierung in verschiedene Datentypen

```
Dim strZahl As String
Dim intZahl As Integer
' Zahl vom Typ String
strZahl = "24"
' Konvertierung in einen Integer-Wert
intZahl = Convert.ToInt32(strZahl)
'Rechenoperation mit dem neuen Integer-Wert
intZahl = intZahl * 100
' Ausgabe des Wertes in einer Messagebox
' mit voriger
' Konvertierung in einen String-Wert
MessageBox.Show(intZahl.ToString)
```

Nach der Konvertierung in einen Integer-Wert kann mit diesem gerechnet werden. Um das Ergebnis wieder typengerecht auszugeben, erfolgt die Konvertierung zurück in einen `String`.

In der Tabelle 3.4 finden Sie die jeweiligen Methoden der `Convert`-Klasse, um Datentypen zu konvertieren:

Tabelle 3.4
Konvertierungsmöglichkeiten

Methode	Konvertierung in folgendes Format
Convert.ToString()	String
Convert.ToSingle()	Single
Convert.ToDouble()	Double
Convert.ToBoolean()	Boolean
Convert.ToByte()	Byte
Convert.ToSByte()	SByte
Convert.ToChar()	Char
Convert.ToInt16()	Short
Convert.ToInt32()	Integer
Convert.ToInt64()	Long
Convert.ToUInt16()	UShort
Convert.ToUInt32()	UInteger
Convert.ToUInt64()	ULong
Convert.ToDateTime()	Date
Convert.ToDecimal()	Decimal

Bei einer Umwandlung von einem Datentyp in einen kompatiblen größeren Datentyp (zum Beispiel von `Integer` nach `Long`) brauchen Sie keinen expliziten Code zu schreiben, diese Typumwandlung führt der Compiler automatisch durch. Man spricht in diesem Zusammenhang von einer impliziten Typumwandlung.

3.1.7 Die CType-Methode

Eine weitere Möglichkeit zur Umwandlung von Datentypen bietet Ihnen in Visual Basic 10 die `CType()`-Methode.

Die Syntax der `CType()`-Methode sieht wie folgt aus:

```
CType(Variable, Datentyp)
```

Dabei ist `Variable` ein Platzhalter für die Variable, die den entsprechenden Wert enthält, und `Datentyp` ist der Datentyp, in den die Variable umgewandelt werden soll.

Das folgende Beispiel wandelt eine Variable vom Typ `Object` in ein Objekt vom Typ `Button` um.

```
Private Sub Button1_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles Button1.Click
    Dim b As Button = CType(sender, Button)
End Sub
```

Eine Alternative zu `CType()` ist die `DirectCast()`-Methode, die sehr ähnlich funktioniert.

3.1.8 GetType-Methode

Da es Polymorphie und Vererbung – jeder Datentyp ist von Object abgeleitet – erlauben, dass ein Objekt einen allgemeineren Datentyp besitzen kann als den, mit dem es ursprünglich definiert wurde, ist es oft wichtig, festzustellen, welchen Datentyp eine Variable zur Zeit der Ausführung besitzt. Um sicherzugehen, welchen Datentyp ein Objekt gerade besitzt, kann dieser mit der `GetType()`-Funktion ermittelt werden. Das folgende Beispiel in Listing 3.6 verdeutlicht die Verwendung von `GetType()`. Zunächst wird eine Variable vom Typ `Object` deklariert. Dieser wird ein Wert zugewiesen, anschließend wird in einer `MessageBox` mit `GetType()` der Typ des Objekts ermittelt. Als Ausgabe würde »Der Typ der Variablen ist System.String« erscheinen.

Listing 3.6 Verwendung von GetType()

```
Dim Text As Object
Text = "Dies ist ein Test"
MessageBox.Show("Der Typ der Variable ist " & _
    Text.GetType.ToString)
```

3.1.9 Option Strict On

Standardmäßig erlaubt Visual Basic 10 die Konvertierung eines Datentyps in einen beliebigen anderen. Das kann zu Datenverlust führen, vor allem dann, wenn die Größe des neuen Datentyps überschritten wird. In Visual Basic 6 war es beispielsweise möglich, Zahlen aus dem Typ String mit Integerzahlen zu addieren. Das ermöglicht zwar eine bequeme und schnelle Programmierung, kann jedoch in größeren Projekten zu schwer auffindbaren Fehlern führen.

In Visual Basic 10 hat sich dieses Problem nicht grundlegend geändert, es gibt jedoch den Modus `Option Strict On`, der implizite Typenumwandlung nicht mehr erlaubt.

Sie sollten `Option Strict On` immer benutzen, um schwerwiegende Folgefehler von vornherein auszuschließen – auch wenn die Entwicklung dadurch länger dauern könnte.

Im Übrigen steht der Visual Basic-Editor dem Programmierer mit konkreten Vorschlägen zur Seite, falls ein Datentyp Gefahr läuft, falsch konvertiert zu werden. Schauen Sie sich das Beispiel in Listing 3.7 einmal näher an:

```
Dim strZahl As String
Dim dblZahl As Double
strZahl = "24"
dblZahl = strZahl * 100
```

Listing 3.7
Unzulässige Rechenoperation mit einem String

Ohne `Option Strict On` gäbe der Editor keine Meldung aus, die Rechenoperation würde ohne weitere Prüfung durchgeführt werden. Bei größeren Zahlen, mit denen noch dazu umfangreiche Berechnungen angestellt würden, ist die Gefahr des Datenverlusts jedoch nicht auszuschließen, was fehlerhafte Ausgaben zur Folge hätte.

Wenn wir nun `Option Strict On` hinzuschalten, was Sie eigentlich immer tun sollten, würden wir einen Compiler-Fehler angezeigt bekommen.

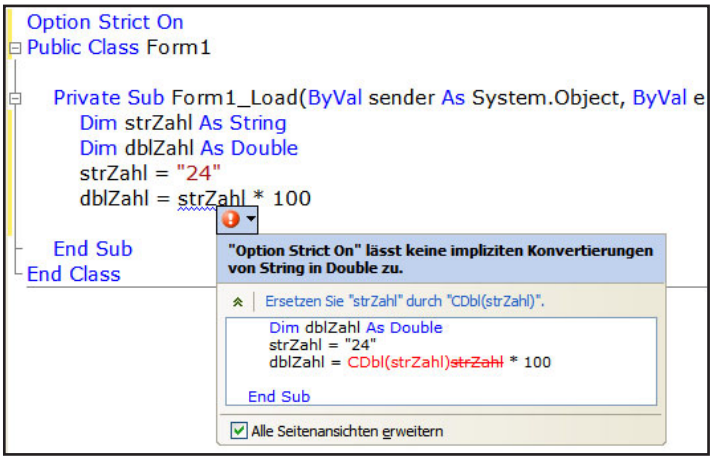


Abbildung 3.2
Meldungsbox bei unzulässiger Verwendung von Variablen bei der Compiler-Einstellung Option Strict On

Wie Sie sehen, unterbreitet Ihnen Visual Studio gleich einen Vorschlag, wie Sie das Problem lösen können. Mit einem Klick wird `strZahl` zu `CDbl(strZahl)` und Sie können mit der Arbeit an Ihrem Programm fortfahren.

Tipp

`Option Strict On` können Sie in Visual Studio 2010 in Ihren Projekteigenschaften festlegen. Für jedes neue Projekt, das Sie beginnen, müssen Sie diese Eigenschaft erneut festlegen oder in der Projektvorlage diese Eigenschaft als Standard für jedes neue Projekt setzen. Sie können ebenfalls an oberster Stelle der Codedatei folgende Anweisung hinzufügen: `Option Strict On`.

3.1.10 Programmiernotation

Gerade für Fortgeschrittene stellt sich oft die Frage, wie Objekte, wie zum Beispiel Textfelder, Group-Boxen, Buttons etc., oder auch Variablen und Arrays richtig benannt werden sollen. Viele Programmierer benennen Objekte und Variablen nach Belieben, wie es ihnen gerade in den Sinn kommt. Bei kleinen Projekten und Programmen mag das vielleicht nicht weiter tragisch sein, wenn man jedoch eine größere Applikation bereitstellen oder im Team an einem Projekt arbeiten möchte, wird diese Willkür schnell zum Albtraum. Da kann es vorkommen, dass man selbst nicht mehr weiß, wofür die Variable jetzt eigentlich gedacht war oder welcher Inhalt in die seltsam benannte Textbox kommt. Schlimmer kann es dann noch werden, wenn man selbst nicht mehr weiß, ob es sich bei dem benutzten Namen um eine Variable oder um ein Objekt handelt. So kann es schnell passieren, dass man einer Variablen einen falschen Wert zuweist oder eine Textbox mit falschen Daten versorgt. Lesen Sie die nächsten Zeilen bitte mit besonderer Sorgfalt. Wenn Sie sich die richtige Namensgebung gleich zu Beginn angewöhnen, werden Sie später keine Sorgen mit den oben geschilderten Problemen bekommen.

Vielleicht haben Sie schon etwas von der ungarischen Konvention gehört. Diese wurde von Charles Simonyi für Programmiersprachen vor der .NET-Zeit eingeführt. Dabei wurde der Gültigkeitsbereich der Variablen sowie der Variablentyp als Präfix geschrieben.

`g_iNummer` war zum Beispiel eine `Integer`-Variable, die global deklariert war.

Mit .NET hat Microsoft sich aber von diesem Standard entfernt und einige Neuerungen im Bereich der Framework-Programmierung hinzugefügt.

Wie Sie vielleicht wissen, ist ein Namespace eine logische Dateneinheit, die für die Organisation von Klassen verwendet wird. Die allgemein gültige Konvention für die Namensgebung von Namespaces ist, den Namen des Unternehmens, von dem der Namespace bereitgestellt wird, an die erste Stelle zu setzen. An zweiter Stelle folgt dann der technologische Name. Die Bereiche werden jeweils mit einem Punkt voneinander getrennt. Nehmen wir zum Beispiel an, ein Unternehmen mit dem Namen *DataService* hat eine Datenbibliothek bereitgestellt. Der richtige Namespace-Name könnte dann Namespace `DataService.Data` lauten.

Sie sollten für Ihre Klassen möglichst »sprechende« Bezeichner vergeben. Außerdem ist zu empfehlen, jeden Anfangsbuchstaben eines neuen Worts groß zu schreiben (Pascal-Notation). Unterstriche sollten vermieden werden. Ein Name für eine Klasse könnte zum Beispiel `StartProcedure` lauten. Ausgehend von unserem oberen Beispiel hieße die Klasse dann `DataService.Data.StartProcedure`. Definiert würde sie im Namespace dann beispielsweise mit `Public Class StartProcedure`.

Für Interfaces sollte ebenfalls ein Bezeichner verwendet werden, der beschreibt, wofür das Interface benötigt wird. Um deutlich zu machen, dass es sich um ein Interface handelt, sollten Interface-Namen immer mit einem großen I beginnen.

```
Interface ICall
```

Als Methode bezeichnet man eine Aktion oder ein Verhalten, das von einem Objekt ausgeführt wird. Hier sollte als Bezeichnung ein Verb verwendet werden. Methoden sollten auch in der Pascal-Notation geschrieben werden.

```
Public Function ShowAll() As Boolean
```

Parameter sollten ebenfalls selbst erklärend sein. Der erste Buchstabe sollte kleingeschrieben werden und jedes weitere Wort im Bezeichner sollte mit einem Großbuchstaben beginnen (camelCasing-Notation).

```
Public Function ShowAll( _
    ByVal firstName As String, _
    ByVal lastName As String)
```

Ein Event wird von einem Objekt als eine Benachrichtigung gesendet. Diese Benachrichtigung kann daraufhin weiterverarbeitet werden (in Kapitel 5 erfahren Sie dazu mehr). Ein EventHandler sollte immer das Suffix `EventHandler` tragen, damit man ihn als solchen identifizieren kann, und in der Pascal-Notation geschrieben sein.

```
Public Delegate Sub ObjectEventHandler
```

Ein Suffix sollte ebenso jedem Attribut mit angehängt werden.

```
Public Class ShowAllAttribute
```

Private Felder, sowie auch lokale Variablen, sollten in der camelCasing-Notation geschrieben werden, während für Eigenschaften die Pascal-Notation zu verwenden ist.

```
Private vorname As String
```

3.1.11 Implizite Typisierung lokaler Variablen

Unter impliziter Typisierung von lokalen Variablen versteht man, dass eine Variable nicht mehr mit einem konkreten Datentyp definiert werden muss, wenn sie bei der Definition einen Wert zugewiesen bekommt.

```
Dim value = 17
```

Der Compiler erkennt aufgrund der Wertzuweisung, dass es sich hierbei um eine Variable vom Typ `Integer` handelt, und führt die Typisierung implizit durch.

Achtung

Bitte denken Sie jetzt keinesfalls, dass nun der Datentyp `Variant` wieder eingeführt worden ist. Implizite Typisierung vergibt den Datentyp automatisch, deswegen muss auch die Zuweisung eines Werts zwingend bei der Definition der Variablen geschehen. Eine Änderung des Datentyps im weiteren Verlauf der Methode, wie es bei `Variant` ja möglich war, ist nicht erlaubt.

Auch IntelliSense unterstützt Sie in Bezug auf implizit typisierte Variablen wie gewohnt, wie Sie in Abbildung 3.3 sehen.

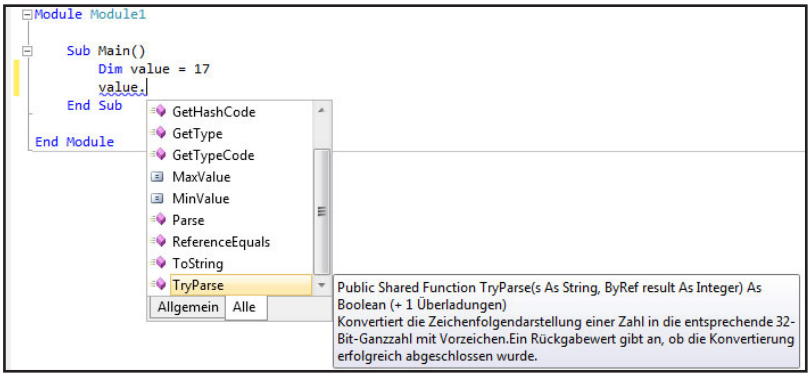


Abbildung 3.3
IntelliSense bei impliziter Typisierung

Implizite Typisierung ist jedoch nur für lokale Variablen erlaubt und eine Zuweisung mit `Nothing` ist ebenfalls nicht gestattet.

Die Variable muss bei der Definition auch initialisiert werden, wobei eine Initialisierung auch durch einen Funktionsaufruf erfolgen kann.

Wenn Sie sich die Mühe machen und einen Blick in den IL-Code werfen – das Tool *ildasm.exe* eignet sich hervorragend dazu – werden Sie sehen, dass die Variable tatsächlich auch als `Integer` definiert ist.

Ich möchte jedoch gleich vorausschicken, dass Sie implizite Typisierung nicht bei so offensichtlichen Definitionen anwenden sollten wie in diesem kleinen Beispiel. Das würde mit Sicherheit nur die Lesbarkeit Ihres Programmcodes verschlechtern und nicht wirklich einen Vorteil bringen.

Sie werden jedoch im Verlauf dieses Buchs noch sehen, dass es aufgrund von LINQ Situationen gibt, bei denen Sie den Rückgabewert einer Funktion gar nicht kennen und unter Umständen auch gar nicht definiert haben. Und genau für solche Fälle wurde die implizite Typisierung eingeführt.

3.2 Kontrollstrukturen

Wie Ihnen bekannt sein wird, werden Befehle in Visual Basic grundsätzlich von oben nach unten, das heißt, Zeile für Zeile abgearbeitet. Der Compiler führt jede Zeile aus und startet erst dann mit der nächsten, wenn der vorige

Vorgang abgefertigt ist. Diese Vorgehensweise können Sie mit sogenannten Verzweigungen ändern. Listing 3.8 demonstriert, wie Verzweigungen im eigentlichen Sinne funktionieren.

3.2.1 If-Else-End If

Listing 3.8
Beispiel für Verzweigungen

```
Public Sub Verzweigung()
    Computer = My.Application.Culture.ToString
    Pruefen(Computer)
    MessageBox.Show("Prüfung abgeschlossen")
End Sub
Sub Pruefen(Computer As String)
    If Computer = "de-DE" Then
        MessageBox.Show(
            "Aktuelle Sprache: Deutsch")
    Else
        MessageBox.Show("Sprache nicht bekannt")
    End If
End Sub
```

In Listing 3.8 wird die aktuelle Spracheinstellung des Computers ermittelt und in einen String geschrieben. Danach wird die `Pruefen()`-Methode aufgerufen (in der Programmierung häufig auch als **Invoking** bezeichnet). Dort wird kontrolliert, ob die aktuelle Spracheinstellung Deutsch ist. Falls das der Fall ist, wird eine Meldung ausgegeben. Nachdem die `Pruefen()`-Methode beendet wurde, kehrt die Programmausführung wieder in die aufrufende Methode zurück und führt dort die nächste Zeile aus, die in diesem Beispiel das Programm beendet.

Info

In Listing 3.8 wurde der `My`-Namespace verwendet. Mehr zu diesem Namespace erfahren Sie in Kapitel 5.8, »Das My-Object«.

Der im obigen Beispiel dargestellte Else-Zweig ist optional. Genauso können Sie mit `ElseIf`- weitere Else-Zweige mit unterschiedlichen Bedingungen hinzufügen.

3.2.2 Der ternäre Operator IIf

Visual Basic stellt mit dem `IIf`-Operator einen ternären Operator zur Verfügung. Der `IIf`-Operator besitzt drei Argumente. Der erste ist ein zu überprüfender Ausdruck und die anderen beiden Argumente sind mögliche Werte, die den Rückgabewert dieses Operators festlegen. Ist der erste Ausdruck wahr, dann wird das zweite Argument zurückgegeben, ansonsten das dritte Argument. Im nachfolgenden Beispiel wird überprüft, ob eine Variable `i` einen Wert größer als 5 besitzt. Ist das der Fall, wird das zweite Argument zurückgegeben, ansonsten das dritte Argument.

```
Dim s As String = IIf(i>5, "groß", "klein").ToString()
```

NEW

Beim `IIf`-Operator wird seit Visual Basic 9 nur das Argument ausgewertet, das tatsächlich auch zurückgegeben wird. In den Vorgängerversionen wurden immer beide möglichen Rückgabewerte ausgewertet.

Die Anweisung

```
Dim i as Integer = Convert.ToInt32(IIf (x Is Nothing, 0,
x.Test))
```

hätte früher, falls x tatsächlich `Nothing` wäre, eine `NullReferenceException` als Laufzeitfehler geworfen, weil die Eigenschaft `Test` (das zugehörige Objekt ist ja `Nothing`) nicht ausgewertet werden konnte. Jetzt wird einfach 0 zurückgegeben.

3.2.3 Select Case

Wenn Sie einen festen Ausdruck mit verschiedenen Alternativen vergleichen wollen, kann es sehr umständlich werden, diesen mit einer Aneinanderreihung von `If`-Ausdrücken (oder auch `ElseIf`-Ausdrücken) zu bewältigen. Praktischer und einfacher geht es mit dem `Select Case`-Ausdruck. Dieser überprüft eine Variable, deren Wert mit den Werten der einzelnen Case-Zweige verglichen wird. Jeder Case-Zweig besitzt einen Anweisungsblock. Bei der Programmausführung kommt derjenige zum Zug, dessen Wert mit dem der Variablen übereinstimmt. In Listing 3.9 sehen Sie, wie eine `Select Case`-Anweisung aufgebaut ist:

Listing 3.9 Select Case-Anweisung

```
Select Case ZuPrüfendeVariable
    Case Wert1
        'Befehl
    Case Wert2
        'Befehl
    Case Wert2
        'Befehl
    Case Else
        'Befehl
End Select
```

Der `Case Else`-Block wird ausgeführt, wenn in der vorhergehenden Liste kein Wert auf den zu prüfenden Wert passt. Der `Case Else`-Block ist optional.

Dabei ist es auch möglich, statt eines konkreten Werts einen Wertebereich anzugeben, wie in Listing 3.10 dargestellt.

Listing 3.10 Select Case-Anweisung mit Werteblöcken

```
Select Case ZuPrüfendeVariable
    Case 1 To 5
        'Befehl
    Case 6 To 20
        'Befehl
    Case 21 To 50
        'Befehl
    Case Else
        'Befehl
End Select
```

3.3 Schleifen

Schleifen dienen in der Regel dazu, Befehle so lange zu wiederholen, bis eine bestimmte Bedingung eingetreten ist. Dieser Vorgang wird zu Beginn oder am Ende der Schleife festgelegt und bei jedem Schleifendurchlauf am Anfang oder Ende geprüft. Man spricht deswegen auch von kopf- und fußgesteuerten Schleifen.

3.3.1 Do Loop-Schleife

Die `Do Loop`-Schleife kann in verschiedenen Varianten auftreten. Sie wird dazu verwendet, so lange einen Prozess zu wiederholen, bis eine bestimmte Voraussetzung erfüllt oder nicht mehr erfüllt ist.

Kopfgesteuerte Schleife:

```
Dim i As Integer = 0
Do Until i > 100
    i += Convert.ToInt32(Console.ReadLine())
Loop
```

Fußgesteuerte Schleife:

```
Dim i As Integer = 0
Do
    i += Convert.ToInt32(Console.ReadLine())
Loop Until i > 100
```

Die fußgesteuerte Schleife wird mindestens ein Mal durchlaufen, weil die erste Prüfung erst am Ende der Schleife erfolgt. Die kopfgesteuerte Schleife kann unter Umständen auch kein einziges Mal durchlaufen werden.

3.3.2 Die While-Schleife

Die `While`-Schleife ist sehr ähnlich zur `Do Loop`-Schleife mit dem Unterschied, dass bei der `While`-Schleife die Schleife so lange ausgeführt wird, wie die Bedingung `true` ist.

```
Dim i As Integer
While i < 100
    Console.WriteLine(i)
    i += Convert.ToInt32(Console.ReadLine())
End While
Console.ReadLine()
```

Alternativ können Sie das Schlüsselwort `While` auch in einer `Do Loop`-Schleife verwenden. Statt `Until` wird dann einfach das Schlüsselwort `While` eingesetzt.

```
Dim i As Integer = 0
Do While i < 100
    i += Convert.ToInt32(Console.ReadLine())
Loop
```

3.3.3 Die For Next-Schleife

Wenn Sie bereits wissen, wie oft die Schleife durchlaufen werden soll, empfiehlt sich die `For Next`-Schleife. Bei der Schleife in Listing 3.11 wird die Schleife von 10 bis 100 durchlaufen und der Wert jeweils in der Konsole ausgegeben.

Listing 3.11 For Next-Schleife im Einsatz

```
For i As Integer = 10 To 100
    Console.WriteLine(i)
Next
Console.ReadLine()
```

Optional können Sie auch mit dem Schlüsselwort `Step` die Schrittweite beim Schleifendurchlauf verändern. Standardmäßig ist die Schrittweite +1.

```
For i As Integer = 100 To 10 Step -1
    Console.WriteLine(i)
Next
```

3.3.4 Die For Each-Schleife

Die `For Each`-Schleife wird am häufigsten für das Durchlaufen von Arrays oder Collections benutzt. Hierbei wird nicht wie bei der `For Next`-Schleife ein Zahlenwert hochgezählt, sondern auf ein Element innerhalb einer Liste zurückgegriffen. Die Schleife wird so lange wiederholt, wie Elemente in der Auflistung vorhanden sind.

```
For Each c As Control In Me.Controls
    MessageBox.Show(c.Name)
Next
```

Die `For Each`-Schleife kann über alle aufzählbaren Objekte iterieren.

3.4 Arrays

In Visual Basic 10 gibt es zwei Arten von Arrays. Zum einen Arrays, die einfach mit Klammern deklariert werden. In diesem Fall sind Arrays Variablen, die Werte des gleichen Datentyps speichern können. Zur besseren Veranschaulichung können Sie sich eine Liste vorstellen, die zunächst eine Nummer und dann den dazugehörigen Wert, wie in Tabelle 3.5, enthält.

Tabelle 3.5 Werte in einem Array

Index	Wert
0	Kotz
1	Haban
2	Mühlbauer
3	Klaubert
4	Saucke
5	Graupp
6	Zauner
7	Schuldt

Arrays beginnen – im Gegensatz zur früheren Visual Basic 6-Version – immer mit dem Index 0. Die Index-Obergrenze kann bei der Deklaration in Klammern hinter dem Array-Namen angegeben werden, zum Beispiel:

```
Dim ArrayName(99) As String
```

Hier enthält das Array 100 Werte, da der Index des ersten Elements 0 ist.

Achtung

Zum Vergleich: In C# geben Sie in den Klammern die Anzahl der Elemente an.

Es ist nicht unbedingt erforderlich, die Anzahl der Elemente bei der Deklaration anzugeben. Diese kann auch später durch `ReDim` festgelegt werden, falls die Anzahl der Elemente noch nicht feststeht. Beachten Sie jedoch, dass `ReDim` sehr viel Rechenzeit benötigt.

Im folgenden Listing 3.12 wird gezeigt, wie man ein Array mit Werten füllt und dieses dann in einer Konsole ausgeben kann.

Listing 3.12
Ausgabe von Arrays in einer Konsolenanwendung

```
Sub Main()
    Dim aZahlenbeispiele(9) As Integer
    Dim i As Integer
    aZahlenbeispiele(0) = 1
    aZahlenbeispiele(1) = 2
    aZahlenbeispiele(2) = 3
    aZahlenbeispiele(3) = 4
    aZahlenbeispiele(4) = 5
    aZahlenbeispiele(5) = 6
    aZahlenbeispiele(6) = 7
    aZahlenbeispiele(7) = 8
    aZahlenbeispiele(8) = 9
    aZahlenbeispiele(9) = 10
    For i = 0 To aZahlenbeispiele.Length - 1
        Console.WriteLine(aZahlenbeispiele(i))
    Next
    Console.ReadLine()
End Sub
```

Die Ausgabe würde dann in der integrierten Konsole von Visual Studio 2010 aussehen wie in Abbildung 3.4.

Info

Die Ausgabe der Array-Werte erfolgt hier in einer Konsolenanwendung. Konsolenanwendungen lassen sich mit jeder Visual Studio 2010-Version erstellen und dienen zum Testen der Anwendung in einem Konsolenfenster ähnlich wie unter MS DOS. Eine neue Konsolenanwendung legen Sie wie folgt an: Klicken Sie im Menü auf Datei/Neu/Projekt, und wählen Sie dann unter Visual Basic-Projekte die Vorlage Konsolenanwendung aus. Sie können jetzt innerhalb der *Sub Main*-Prozedur Ihren Code entwickeln. Dieser wird beim Start des Projekts ausgeführt.

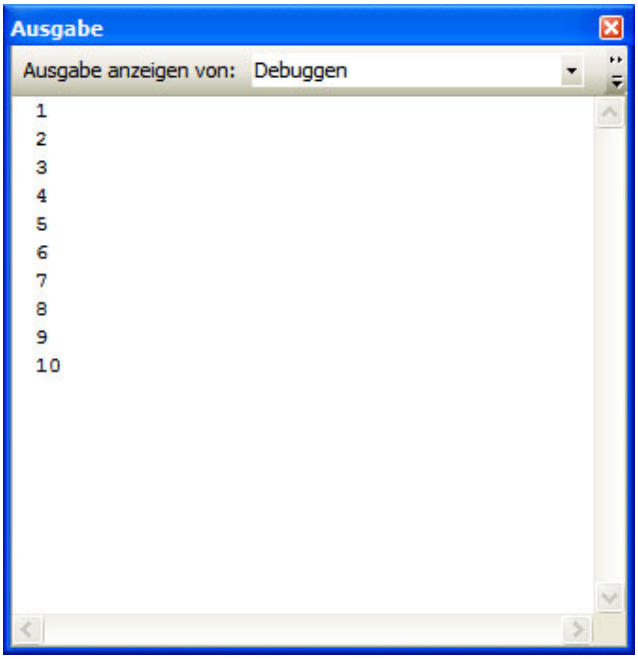


Abbildung 3.4 Ausgabe des Array-Beispiels in der Visual Studio-Konsole

Zum anderen kann ein Array aber auch als Objekt behandelt werden. Hierbei wird das Array bei der Deklaration mit der `CreateInstance`-Methode bekannt gemacht. Die Werte des Arrays müssen mit `SetValue` zugewiesen und mit der `GetValue`-Methode wieder ausgelesen werden. Zur Verdeutlichung wird Listing 3.12 umgeschrieben. Das Ergebnis sehen Sie in Listing 3.13.

Listing 3.13 Ausgabe des Array-Objekts

```
Public Shared Sub Main()
    Dim aZahlenbeispiele As Array = _
        Array.CreateInstance( _
        GetType(Integer), 11)

    aZahlenbeispiele.SetValue(1, 0)
    aZahlenbeispiele.SetValue(2, 1)
    aZahlenbeispiele.SetValue(3, 2)
    aZahlenbeispiele.SetValue(4, 3)
    aZahlenbeispiele.SetValue(5, 4)
    aZahlenbeispiele.SetValue(6, 5)
    aZahlenbeispiele.SetValue(7, 6)
    aZahlenbeispiele.SetValue(8, 7)
    aZahlenbeispiele.SetValue(9, 8)
    aZahlenbeispiele.SetValue(10, 9)
    For i As Integer = 0 To _
        aZahlenbeispiele.Length - 1

        Console.WriteLine _
          (aZahlenbeispiele.GetValue(i) _
           .ToString)
    Next
    Console.ReadLine()
End Sub
```

In der Praxis finden Sie zumeist die erste hier vorgestellte Version von Arrays im Einsatz.

3.4.1 Mehrdimensionale Arrays

Sowohl »gewöhnliche« Arrays als auch Array-Objekte können mehrdimensionale Werte beinhalten. Das folgende Beispiel (Listing 3.14) zeigt ein vierdimensionales Array, das in der ersten Position fünf, in der zweiten zehn, in der dritten 100 und in der vierten wiederum fünf Elemente aufweist. Nach der Vereinbarung (als gewöhnliches Array sowie als Array-Objekt) erfolgt in der nächsten Zeile jeweils die Wertzuweisung an ein einzelnes Element.

Listing 3.14
Mehrdimensionale Arrays

```
Public Shared Sub Main()
    Dim Werte(4, 9, 99, 4) As Integer
    Werte(1, 2, 3, 4) = 500
    'Oder mit dem Array-Objekt
    Dim NeueWerte As Array = _
    Array.CreateInstance(GetType(Integer), _
        4, 9, 99, 4)
    NeueWerte.SetValue(500, 1, 2, 3, 4)
End Sub
```

Das oben definierte Array kann im Übrigen 25.000 Elemente aufnehmen (5*10*100*5).

3.4.2 Arrays hinzufügen

Oftmals wissen Sie während der Programmierung nicht, wie viele Werte ein Array später enthalten wird (stellen Sie sich einmal vor, Sie lesen den gesamten Inhalt eines Verzeichnisses in ein Array, können aber vorher nicht bestimmen, wie viele Dateien in diesem Verzeichnis abgelegt sind). Hier schafft die `ReDim`-Anweisung Abhilfe.

Um die bereits gespeicherten Werte nicht zu verlieren, sollten Sie dabei das Schlüsselwort `Preserve` wie in Listing 3.15 benutzen.

Listing 3.15
Arrays mit ReDim erweitern

```
Public Sub Addmore()
    Dim aWerte(9) As Integer
    For i As Integer = 0 To 9
        aWerte(i) = i + 1
    Next i
    'ReDim, Werte bleiben erhalten
    ReDim Preserve aWerte(19)
    For i As Integer = 10 To 19
        aWerte(i) = i + 1
    Next i

    For i As Integer = 0 To aWerte.Length - 1
        Console.WriteLine(aWerte(i))
    Next
    Console.ReadLine()
End Sub
```

Tipp

Ohne dem weiteren Verlauf dieses Buchs vorgreifen zu wollen, möchte ich darauf hinweisen, dass bei so einer Aufgabenstellung Collections die bessere Wahl sind.

3.4.3 Arrays sortieren

Mit der `Array.Sort()`-Methode können Sie die Werte eines Arrays sortieren. Die Methode vergleicht dabei jeden einzelnen *Array*-Wert mit den anderen. Das Beispiel in Listing 3.16 erstellt mit der `Random`-Klasse zehn zufällige Werte zwischen 1 und 100 und legt diese Werte in dem Array `aWerte` ab. Danach werden die unsortierten Werte ausgegeben, bevor sie sortiert und dann erneut ausgegeben werden.

Listing 3.16 Arrays sortieren

```
Public Sub Sortieren()
    Dim aWerte(9) As Integer
    Dim myrand As New Random()
    For i As Integer = 0 To 9
        aWerte(i) = myrand.Next(1, 100)
    Next i
    'Unsortierte Werte ausgeben
    Console.WriteLine("Unsortierte Werte:")
    For i As Integer = 0 To aWerte.Length - 1
        Console.WriteLine(aWerte(i))
    Next
    Console.WriteLine(vbNewLine & _
        "Nun folgen die sortierten Werte:")
    'Array sortieren
    Array.Sort(aWerte)
    For i As Integer = 0 To aWerte.Length - 1
        Console.WriteLine(aWerte(i))
    Next
End Sub
```

Die Ausgabe könnte so aussehen:

```
Unsortierte Werte:
11
64
8
74
47
10
78
52
66
51
Nun folgen die sortierten Werte:
8
10
11
47
51
52
64
66
74
78
```

3.4.4 Arrays invertieren

Mit der `Array.Reverse()`-Methode können Sie geordnete Arrays in umgekehrter Reihenfolge ordnen. Das Beispiel in Listing 3.17 zeigt, wie Sie einem Array zunächst Werte von 1 bis 10 in aufsteigender Reihenfolge hinzufügen und diese dann umkehren.

Listing 3.17
Arrays umkehren

```
Public Sub Reverse()
    Dim aWerte(9) As Integer
    For i As Integer = 0 To 9
        aWerte(i) = i + 1
    Next i
    Array.Reverse(aWerte)
    For i As Integer = 0 To aWerte.Length - 1
        Console.WriteLine(aWerte(i))
    Next
    Console.ReadLine()
End Sub
```

3.4.5 Arrays durchsuchen

Nachdem das Array nun sortiert wurde, kann dieses auch durchsucht werden. Der Suchalgorithmus arbeitet auch noch bei großen Arrays sehr schnell und bietet so die optimale Möglichkeit, nach bestimmten Daten zu suchen. Nachdem die Werte in dem Array abgelegt wurden, müssen diese mit der `Array.Sort()`-Methode sortiert werden, danach kann über `Array.BinarySearch()`der Index der zu suchenden Person ermittelt werden. Am Ende der Beispielanwendung in Listing 3.18 werden der neue Index nach der Sortierung und der Name der Person ausgegeben.

Listing 3.18
Array nach bestimmtem Wert durchsuchen

```
Public Sub Suchen()
    Dim aPersonen(7) As String
    Dim zuSuchendePerson As String
    Dim Result As String
    Dim personIndex As Integer
    aPersonen(0) = "Kotz"
    aPersonen(1) = "Haban"
    aPersonen(2) = "Klaubert"
    aPersonen(3) = "Mühlbauer"
    aPersonen(4) = "Graup"
    aPersonen(5) = "Bremer"
    aPersonen(6) = "Zauner"
    aPersonen(7) = "Albert"
    Array.Sort(aPersonen)
    zuSuchEndePerson = "Graup"
    personIndex = Array.BinarySearch(aPersonen, _
        zuSuchendePerson)
    Result = aPersonen(personIndex)
    MessageBox.Show _
        ("Die zu suchende Person wurde im " & _
         "Index an der neuen Stelle " & _
         personIndex & _
         " gefunden. Die Person ist: " & Result, _
         "Suche in Arrays", _
         MessageBoxButtons.OK, _
         MessageBoxIcon.Information)
End Sub
```

Die Ausgabe sehen Sie in Abbildung 3.5.

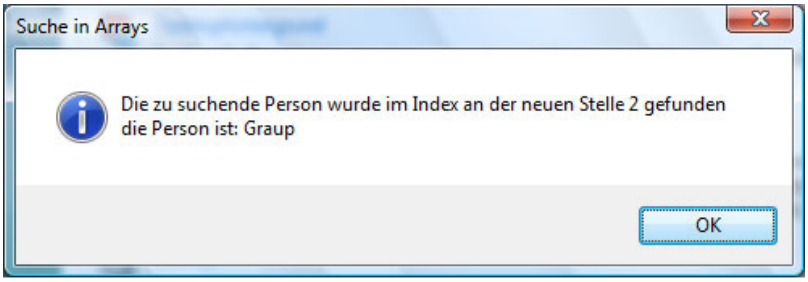


Abbildung 3.5
Ergebnis der Array-Durchsuchung

3.5 Operatoren

Operatoren verknüpfen im eigentlichen Sinn zwei Ausdrücke miteinander. Diese Ausdrücke können aus Zeichenketten (Strings), Zahlen oder Formeln bestehen. Bei der Programmierung unterscheiden wir grundsätzlich unterschiedliche Arten von Operatoren.

- Arithmetische Operatoren dienen dazu, Berechnungen durchzuführen.
- Verknüpfungsoperatoren verknüpfen unterschiedliche Ausdrücke miteinander.
- Logische Operatoren werden zur bitweisen Verknüpfung von Operanden benötigt.
- Vergleichsoperatoren vergleichen unterschiedliche Ausdrücke miteinander.

Die arithmetischen Operatoren in Tabelle 3.6 können bei der Programmierung mit Visual Basic 10 verwendet werden:

Tabelle 3.6
Operatoren im Überblick

Arithmetischer Operator	Funktion
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
\	Integerdivision
^	Potenzierung
Mod	Bestimmung des Restwerts

Die Darstellung der Grundrechenarten ist in Visual Basic denkbar einfach. Hier einige Beispiele für die Verwendung von Grundrechenarten:

Beispiel für Addition:

```
dblAusgabe = 10 + 3
```

Beispiel für Subtraktion:

```
dblAusgabe = 10 - 3
```

Beispiel für Multiplikation:

```
dblAusgabe = 10 * 3
```

Beispiel für Division:

```
dblAusgabe = 10 / 3
```

Die Integerdivision liefert im Gegensatz zur Division keine gebrochenen Zahlen, sondern einen ganzzahligen Wert.

Achtung

Seien Sie bitte nicht überrascht, dass es bei einer Gleitkommazahlendivision (mit /) bei der Division durch 0 keinen Fehler gibt. Es wird mit einer Annäherung von 0 dividiert.

Anstelle des Operators / wird der Backslash \ eingesetzt.

```
dblAusgabe = 10 \ 3
```

Im Gegensatz zur Integerdivision liefert der Mod-Operator den Rest einer Division.

```
dblAusgabe = 10 Mod 3
```

Bei der Potenzierung wird nicht 10^3 geschrieben, sondern das Zeichen ^ verwendet.

```
dblAusgabe = 10 ^ 3
```

Visual Basic kennt seit der Sprachversion 8, die mit Visual Studio 2005 veröffentlicht wurde, auch zusammengesetzte Zuweisungsoperatoren. Anstatt beim Hochzählen einer Variablen beispielsweise `i = i + 1` zu schreiben, können Sie den Shortcut `i += 1` verwenden. Dasselbe gilt für den Subtraktions-, Multiplikations- und Divisionsoperator.

Verknüpfungsoperatoren werden immer dann gebraucht, wenn Sie in einer Ausgabe einen String mit einem anderen String oder einem anderen Datentyp verwenden wollen. Für die Verbindung wird das Kaufmannsund (&) als Verknüpfungsoperator benötigt.

```
MessageBox.Show("Der Wert der Zahl beträgt " & _
    intZahl1.ToString)
```

Logische Operatoren dienen dazu, mehrere Teilbedingungen miteinander zu verknüpfen und damit zum Beispiel Wahrheitsüberprüfungen festzustellen.

Visual Basic kennt die in Tabelle 3.7 aufgelisteten logischen Operatoren.

Tabelle 3.7
Logische Operatoren

Operator	Funktion
And	Und-Verknüpfung
Or	Oder-Verknüpfung
Not	Negation

Tabelle 3.7 (Forts.) Logische Operatoren

Operator	Funktion
Xor	Exklusive Oder-Verknüpfung
AndAlso	Und-Verknüpfung, die jedoch die Überprüfung aller Ausdrücke abbricht, falls das Ergebnis eindeutig ist
OrElse	Oder-Verknüpfung, die jedoch die Überprüfung aller Ausdrücke abbricht, falls das Ergebnis eindeutig ist

Beispiel für den `And`-Operator:

```
If intZahl1 > 1 And intZahl2 < 100 Then
MessageBox.Show("Der gesuchte Wert liegt zwischen 1 und 100")
```

Beispiel für den `Or`-Operator:

```
If intZahl1 < 1 Or intZahl2 > 100 Then
MessageBox.Show("Der Wert liegt unter 1 oder über 100")
```

Der `Xor`-Operator stellt eine Besonderheit dar. Bei `Xor` wird ein logischer Ausschluss zweier boolescher oder ein bitweiser Ausschluss zweier numerischer Ausdrücke durchsucht.

```
If (intZahl1 = 2 Xor 8) > 0 Then
MessageBox.Show("Der Ausdruck ist wahr").
```

Der `Not`-Operator kehrt den Wert des Ausdrucks um. Wenn also beispielsweise das Ergebnis einer If-Abfrage `True` ist, ändert der `Not`-Operator den Wert zu `False`, andersherum ändert er den `False`-Wert zu `True`.

Vergleichsoperatoren werden häufig bei `If`-Anweisungen benutzt, um Variablen und/oder Werte miteinander zu vergleichen.

Tabelle 3.8 zeigt die in Visual Basic 9 verwendbaren Vergleichsoperatoren.

Tabelle 3.8 Vergleichsoperatoren

Operator	Funktion
=	Gleich
<	Kleiner
>	Größer
<=	Kleiner gleich
>=	Größer gleich
<>	Ungleich

In Listing 3.19 sehen Sie, wie man Zahlenwerte miteinander vergleicht. Hierbei spielt es keine Rolle, ob die Zahl als Wert in einer Variablen gespeichert ist oder direkt in der Anweisung steht.

Listing 3.19 Vergleich von Zahlenwerten

```
Dim intZahl1 As Integer = 5
Dim intZahl2 As Integer = 10
If intZahl1 > intZahl2 Then
    MessageBox.Show(
      "Zahl 1 ist größer als Zahl 2")
ElseIf intZahl1 < intZahl2 Then
```

```
    MessageBox.Show(
      "Zahl 1 ist kleiner als Zahl 2")
ElseIf intZahl1 = intZahl2 Then
    MessageBox.Show(
      "Zahl 1 ist gleich als Zahl 2")
End If
```

Listing 3.19 (Forts.) Vergleich von Zahlenwerten

3.6 Befehle

Komplette Programmanweisungen werden in Visual Basic 10 Befehle genannt. Jeder Befehl endet mit einer neuen Zeile oder einem Doppelpunkt.

```
Dim Text As String = "hallo"
Dim T2 As String = "w" : Dim i As Integer = "1"
```

Hierbei wird der Doppelpunkt oft als unprofessionelle Art der Programmierung gesehen, da es die Lesbarkeit der Anwendung um einiges erschwert.

Falls eine Anweisung sehr lang ist, können Sie im Sinne einer besseren Lesbarkeit den Unterstrich (_) benutzen, um einen Befehl auf mehrere Zeilen zu verteilen:

```
MessageBox.Show("Hallo, was ich schon immer mal …", _
    "Titelüberschrift")
```

Vor dem Unterstrich muss zwingend ein Leerzeichen stehen.

Mit Visual Basic 10 kann nun auf die Verwendung des Unterstrichs verzichtet werden.

3.7 Sonstige Sprachelemente

An dieser Stelle will ich Ihnen noch einen Überblick über weitere wichtige Sprachelemente und Schlüsselwörter geben.

3.7.1 Die Operatoren Is und IsNot

Objektvariablen vergleichen Sie mit den Operatoren `Is` und `IsNot`.

```
If obj1 Is obj2 Then
```

Die obige If-Bedingung ist wahr, wenn beide Variablen auf dasselbe Objekt verweisen.

Wenn es darum geht, zu prüfen, ob eine Objektvariable überhaupt einen Verweis auf ein Objekt enthält, erweist sich der Einsatz des Operators `IsNot` oft als eleganter. Die Codezeile

```
If obj IsNot Nothing Then
```

liest sich vermutlich intuitiver als

```
If Not obj Is Nothing Then.
```

3.7.2 Weitere Schlüsselwörter

Folgende drei Schlüsselwörter, die mit Visual Basic 8 eingeführt worden sind, wollen wir uns hier noch kurz ansehen:

Global

Bevor ich Ihnen das Schlüsselwort `Global` näher erläutere, erlauben Sie mir vorab eine kleine Bemerkung. Ich hoffe nur, dass Sie dieses Schlüsselwort nie benötigen. Ich bin zumindest die letzten fünf Jahre darum herum gekommen.

Denn tatsächlich benötigen Sie es nur, wenn Sie bereits gravierende Namenskonflikte haben. Stellen Sie sich vor, Sie haben eigene Namespaces definiert und wollen diese auch so sprechend wie möglich benennen. Ein sehr löbliches Vorgehen, sei noch kurz angemerkt.

Nun heißt einer von diesen Namespaces zufällig `FirmenName.GlobalClasses.System.Data`.

Und schon haben Sie ein Problem, wenn Sie eine Klasse aus dem Original-Namespace `System.Data` verwenden wollen, denn dieser wird jetzt durch Ihren eigenen Namespace überschattet.

Wenn Sie in Ihrem eigenen Namespace nun folgende Variablendefinition vornehmen:

```
Dim ds As New System.Data.DataSet
```

wird diese nicht funktionieren, da der Compiler eine Klasse `DataSet` in Ihrem eigenen Namespace sucht.

Beheben können Sie das mit dem Schlüsselwort `Global`, indem Sie die Definition folgendermaßen durchführen:

```
Dim ds as New Global.System.Data.DataSet
```

Using

Da wir im .NET Framework ein nichtdeterministisches Verhalten des Garbage Collector haben, können wir nicht vorhersehen, zu welchem Zeitpunkt unsere Objekte wieder im Speicher freigegeben werden. Der Zeitpunkt liegt zwar in der nahen Zukunft, aber unsere Objekte halten so lange noch Zugriff auf Ressourcen, die sie beansprucht haben. Um diese Ressourcen bereits vorzeitig freizugeben, können Sie für Ihre Objekte das `IDisposable`-Interface implementieren und eine Methode `Dispose` bereitstellen, aber Sie können nicht beeinflussen, dass der Entwickler, der Ihre Objekte nutzt, die Methode `Dispose` auch aufruft.

Sie können nun Objekte innerhalb eines `Using`-Blocks definieren. Dadurch wird die Laufzeitumgebung automatisch am Ende des Blocks die entsprechende `Dispose`-Methode aufrufen. Dadurch ist aber auch klar, dass Sie `Using` nur für Objekte benutzen können, die das `IDisposable`-Interface implementieren.

Das Schlüsselwort `Using` verwenden Sie dabei wie im folgenden Beispiel dargestellt:

```
Using (a As New Artikel)
    'beliebiger Code
End Using
```

Eine Verwendung der Objektvariablen ist nach dem Ende des `Using`-Blocks nicht mehr möglich.

Wobei Sie bei dieser Syntaxvariante mehrere Objektvariablen für `Using` angeben können:

```
Using a As New Artikel, b As New Artikel
```

Continue

Das `Continue`-Schlüsselwort können Sie dazu verwenden, bestimmte Anweisungen innerhalb einer Schleife zu überspringen.

Achtung

Verwechseln Sie diese Anweisung bitte nicht mit dem vorzeitigen Beenden der Schleife. Es wird lediglich ans Schleifenende gesprungen und dann die nächste Iteration ausgeführt.

Wenn Sie zum Beispiel Datensätze aus einer Datei oder einem `DataSet` durchlaufen und in Abhängigkeit von einem Wert soll eine Verarbeitung nicht stattfinden, können Sie hier mit `Continue` den verarbeitenden Block überspringen. Es wird dann ans Schleifenende gesprungen und der nächste Datensatz wird verarbeitet. Listing 3.20 zeigt eine Verwendung von `Continue`.

Listing 3.20
Verwenden von Continue

```
For Each x As Artikel In Produkte
    If x.Preis > 50 Then
        Continue For
    End If
    'Ansonsten weiterer Code
Next
```

Das Schlüsselwort `Continue` können Sie für jeden Schleifentyp verwenden.

3.8 Konsolenanwendungen erstellen

Visual Basic 10 ermöglicht nicht nur die Programmierung von grafischen Windows-Anwendungen, sondern auch die von sogenannten Konsolenanwendungen. Diese laufen beim Aufruf in der Eingabeaufforderung (der Windows Command Shell) und führen Ein- und Ausgaben über die Standardgeräte `StdIn` (Tastatur) und `StdOut` (Eingabeaufforderungsfenster) durch und dienen als Grundlage für viele Visual Basic 10-Beispiele, die Sie im Internet finden können. Im Mittelpunkt steht die `Console`-Klasse, die ein Eingabeaufforderungsfenster repräsentiert. Mit dem .NET Framework 2.0 wurde die `Console`-Klasse um zahlreiche Mitglieder erweitert, mit der sich unter anderem die Hintergrund- und die Schriftfarbe und die Größe des Konsolenfensters abfragen und einstellen lassen. Lesen Sie hier, welche Möglichkeiten es bei Konsolenanwendungen gibt und wie Sie diese für Ihre eigenen

Anwendungen nutzen können. Um eine Konsolenanwendung zu erstellen, müssen Sie zunächst ein neues Projekt anlegen, um dann die Vorlage KONSOLENANWENDUNG auszuwählen. Wie Sie sehen können, wird nicht wie gewohnt ein Windows-Formular erzeugt, sondern nur eine *.vb*-Datei mit dem Code-Inhalt, den Sie in Listing 3.21 sehen.

Listing 3.21 Code beim Anlegen einer Konsolenanwendung

```
Module Module1
    Sub Main()
    End Sub
End Module
```

Zwischen der Anweisung `Sub Main()` und `End Sub` spielt sich die gesamte Konsolenanwendung ab. Um einen Text auszugeben, können Sie die Methode `Console.WriteLine()` verwenden:

```
Console.WriteLine("Willkommen in der Anwendung")
Console.WriteLine()
```

Mit der Methode `Console.ReadLine()` können Sie den Anwender zur Eingabe auffordern.

Listing 3.22 Einfache Konsolenanwendung

```
Console.WriteLine("Geben Sie Ihren Namen ein:")
Dim Name As String = Console.ReadLine
Console.WriteLine()
Console.Write("Hallo " & Name & _
    " bitte geben Sie den Titel ein:")
Console.WriteLine()
Console.Title = Console.ReadLine
Console.WriteLine()
Console.WriteLine()
```

Die Hintergrund- und die Schriftfarbe der Konsole lassen sich ebenfalls bestimmen.

Listing 3.23 Setzen von Farben für die Konsole

```
Console.WriteLine("Geben Sie eine Farbe ein " _
    " (z.b. red, green, black):")
Console.WriteLine()
Try
    Dim bgkFarbe As String = Console.ReadLine
    Console.BackgroundColor = _
        CType(System.Enum.Parse( _
        GetType(ConsoleColor), _
        bgkFarbe, True), ConsoleColor)
Catch ex As Exception
    Console.Write _
        ("Farbe konnte nicht gefunden werden")
End Try
Console.Write( _
    "Geben Sie die Farbe der Schrift ein:")
Try
    Dim fontFarbe As String = Console.ReadLine
    Console.ForegroundColor = _
        CType(System.Enum.Parse( _
            GetType(ConsoleColor), _
            fontFarbe, True), ConsoleColor)
Catch ex As Exception
    Console.Write _
        ("Farbe konnte nicht gefunden werden")
End Try
```

Die Größe des Konsolenfensters können Sie mit `Console.SetWindowSize(x, y)` festlegen. Zudem können Sie noch die Position des Cursors bestimmen und ihn auf Wunsch verbergen.

```
Console.WriteLine("Cursor unsichtbar machen")
Console.CursorVisible = False
Console.WriteLine()
Console.WriteLine("Weiter geht es mit Enter")
Console.ReadLine()
Console.CursorVisible = True
```

Die Konsole ist neuerdings auch musikalisch (okay, zugegeben, nur der interne Systemlautsprecher kann benutzt werden).

Listing 3.24
Musik in der Konsole

```
Dim Frequenz As Integer
Console.Write("Bitte Frequenz eingeben, " & _
    diese sollte zwischen 37 und 32767 liegen")
Frequenz = Convert.ToInt32 (Console.ReadLine)
Console.WriteLine()
Dim Länge As Integer
Console.Write("Geben Sie die Abspiellänge ein: ")
Länge = Convert.ToInt32(Console.ReadLine)
Console.Beep(Frequenz, Länge)
Console.WriteLine()
Console.WriteLine("Enter beendet die Demo")
Console.ReadLine()
```

Zwar sieht das Ganze nach Spielerei aus, die Konsole kann Ihnen aber bei der alltäglichen Programmierarbeit praktisch zur Seite stehen. Sie können beispielsweise Codefragmente bzw. die Ausgabe von Daten einfach in der Konsole testen und müssen dafür nicht extra Ihre gesamte Anwendung kompilieren.

Außerdem werden Konsolenanwendungen auch sehr gerne bei kleinen Programmen für administrative Aufgaben eingesetzt.

4

Ausnahmebehandlung

Mit der Einführung des .NET Framework Anfang des Jahrtausends wurde für die Visual Basic-Programmierer eine neue Art und Weise der Fehlerbehandlung eingeführt. Der Programmierer muss sich nun nicht mehr um Fehler, sondern lediglich mehr um Ausnahmen kümmern. Das Wort `Exception` trifft dabei auch die Bedeutung besser, denn Ausnahmen sind nicht immer zwangsläufig auch Fehler. Diese neue Art der Ausnahmebehandlung wird auch strukturierte Ausnahmebehandlung genannt und sollte auf jeden Fall Ihre Wahl bei der Ausnahmebehandlung sein. Aus Abwärtskompatibilitätsgründen wird jedoch auch weiterhin die unstrukturierte Fehlerbehandlung im neuen Visual Basic unterstützt.

Sie haben grundsätzlich die Möglichkeit, in Ihrem Code beide Arten der Ausnahmebehandlung einzusetzen, innerhalb einer Methode muss die Art der Ausnahmebehandlung jedoch eindeutig sein.

4.1 Strukturierte Ausnahmebehandlung

Strukturierte Ausnahmebehandlung, oder auch **Structured Exceptionhandling**, ist eine einfache und übersichtlich strukturierte Möglichkeit, auf Programmausnahmen zu reagieren. Diese Art der Ausnahmenbehandlung ist innerhalb des .NET Framework in allen Programmiersprachen implementiert, um eine sprachübergreifende saubere Möglichkeit zu haben, auf vorhersehbare oder auch unvorhersehbare Programmausnahmen zu reagieren.

Die wichtigsten Schlüsselwörter beim Abfangen einer Ausnahme sind dabei `Try`, `Catch` und `Finally` sowie `Throw`, um eine eigene Ausnahme auszulösen.

4.1.1 Abfangen einer Ausnahme

Für das Abfangen einer Ausnahme benötigen Sie zumindest die beiden Schlüsselwörter `Try` und `Catch`, wogegen `Finally` optional verwendet werden kann.

Mit `Try` leiten Sie eine Ausnahmebehandlung ein, das Konstrukt wird mit `End Try` abgeschlossen. Der Code zwischen `Try` und `Catch` kennzeichnet den ersten Block. Wird nun innerhalb dieses Blocks eine Ausnahme ausgelöst, oder besser gesagt geworfen, dann wird der Code im `Catch`-Block ausgeführt. Dabei ist es möglich, für unterschiedliche Ausnahmen auch unterschiedliche `Catch`-Blöcke zu definieren. Wird innerhalb des `Try`-Blocks keine Ausnahme geworfen, so wird der Code ausgeführt, der nach dem Konstrukt implementiert wurde.

Im allereinfachsten Fall besteht eine Ausnahmebehandlung aus zwei Blöcken, dem `Try`-Block und einem `Catch`-Block.

```
Try
    'mach etwas
Catch
    'mach hier deine Ausnahmebehandlung
End Try
```

Sie können jedoch auch mehrere `Catch`-Blöcke definieren, um auf verschiedene Ausnahmen auch unterschiedlich reagieren zu können. Dazu definieren Sie für jede denkbare Ausnahme einen eigenen `Catch`-Block, indem Sie eine Typdefinition hinter dem `Catch` schreiben. Dieser Block behandelt dann eine Ausnahme, die dem angegebenen Typ oder einem von diesem vererbten Typ entspricht. Wird keine Typdefinition angegeben, wird jede Ausnahme, die vom Typ `Exception` abgeleitet ist, innerhalb dieses Blocks behandelt. Da sämtliche Ausnahmentypen von dem Typ `Exception` abgeleitet sind, sollte innerhalb Ihrer Ausnahmebehandlung mindestens ein `Catch`-Block für den allgemeinen Typ `Exception` vorhanden sein. Wenn Sie nämlich nur spezielle Ausnahmen in `Catch`-Blöcken abfangen, dann tritt bei einer unvorhergesehenen Ausnahme ein Programmabbruch auf, da dieser von keinem `Catch`-Block behandelt wird. Der `Exception-Catch`-Block sollte jedoch immer der letzte sein, da ansonsten nachfolgende spezielle Ausnahmebehandlungen ignoriert würden, weil die vorher angegebene allgemeine Ausnahmebehandlung durchgeführt wird.

Das Beispiel aus Listing 4.1 stellt auf einer Windows Form zwei Textfelder zur Eingabe von Zahlen bereit. Diese sollen, ausgelöst durch einen Button-Klick, dividiert und das Ergebnis soll in einem Meldungsfenster ausgegeben werden.

Was kann in diesem Miniprogramm alles schief gehen? Zum einen besteht die Möglichkeit, dass keine Zahlen eingegeben werden, und zum Zweiten könnte für den Nenner die Zahl 0 eingegeben werden. Beides würde zu einer Ausnahme führen, die wir in unterschiedlichen `Catch`-Blöcken behandeln.

Listing 4.1 Einfaches Beispiel für strukturierte Ausnahmebehandlung

```
Private Sub btnCalculate_Click( _
    ByVal sender As Object, _
    ByVal e As System.EventArgs) _
    Handles btnCalculate.Click
```

Listing 4.1 (Forts.) Einfaches Beispiel für strukturierte Ausnahmebehandlung

```
    Try
        Dim Ergebnis As Double
        Ergebnis = _
            Convert.ToDouble(txtZaehler.Text) / _
            Convert.ToDouble(txtNenner.Text)
        MessageBox.Show(Ergebnis.ToString())
    Catch ex As FormatException
        txtZaehler.Text = 1
        txtNenner.Text = 1
        MessageBox.Show(ex.Message & _
            "Geben Sie Zahlen ein, und " & _
            "versuchen Sie es noch einmal")
    Catch ex As DivideByZeroException
        txtNenner.Text = 1
        MessageBox.Show(ex.Message)
    Catch ex As Exception
        MessageBox.Show(ex.Message, _
            "Unerwartete Ausnahme")
    End Try
End Sub
```

In diesem Beispiel wird in der Ausnahmebehandlung die Ausnahmemeldung durch die Eigenschaft `Message` ausgegeben. Das `Exception`-Objekt, das hier mittels der Objektvariablen `ex` angesprochen werden kann, bietet aber auch noch andere Möglichkeiten. Für die Anzeige in einer Oberfläche eignet sich dabei die oben beschriebene Eigenschaft `Message`, aber für Fehlerprotokolle oder Einträge in das Ereignisprotokoll könnte man auch die Eigenschaft `StackTrace` verwenden, die mehr Informationen über die geworfene Ausnahme enthält. Im `StackTrace` ist unter anderem auch die Zeilennummer der Anweisung enthalten, welche die Ausnahme auslöste, sowie sämtliche Methoden, die den Aufruf verursachten.

Im .NET-Framework gibt es seit der Version 2.0 die Möglichkeit, einen `Catch`-Block mit einer zusätzlichen Bedingung zu verknüpfen. Diese Bedingung kann mit dem Schlüsselwort `When` hinter der Typdefinition angegeben werden. Ergibt die Bedingung den Wert `True`, so wird der Block ausgeführt, ansonsten natürlich nicht.

Damit kann man jetzt für ein und denselben Fehler in Abhängigkeit von bestimmten Bedingungen unterschiedliche Ausnahmebehandlungen ansteuern. Das folgende Beispiel zieht eine boolesche Variable zur Prüfung des EventLog heran.

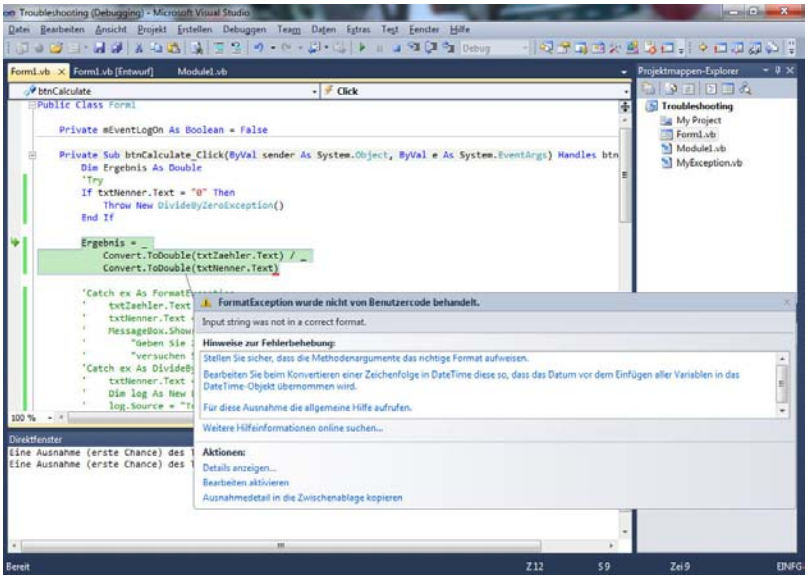
```
Catch ex As DivideByZeroException When _ mEventLogOn = True
    'schreibe ins EventLog...
Catch ex As DivideByZeroException When _ mEventLogOn = False
    MessageBox.Show(ex.Message)
```

Sie können natürlich auch mittels `When` steuern, ob eine spezifische Ausnahme oder der allgemeine `Catch`-Block ausgelöst wird.

Wenn Sie sich nicht sicher sind, von welchem Typ eine bestimmte Ausnahme ist, dann provozieren Sie einfach innerhalb der Entwicklungsumgebung die entsprechende Ausnahme.

Hierzu habe ich alle `Catch`-Blöcke aus Listing 4.1 einkommentiert und in die beiden Textboxen keine Eingabe vorgenommen, um eine unbehandelte Ausnahme auszulösen. In Abbildung 4.1 sehen Sie, welche Tipps Ihnen Visual Studio 2010 gibt, um diese Art der Ausnahme zu vermeiden, und welche Informationen dabei der Entwickler zur Verfügung gestellt bekommt.

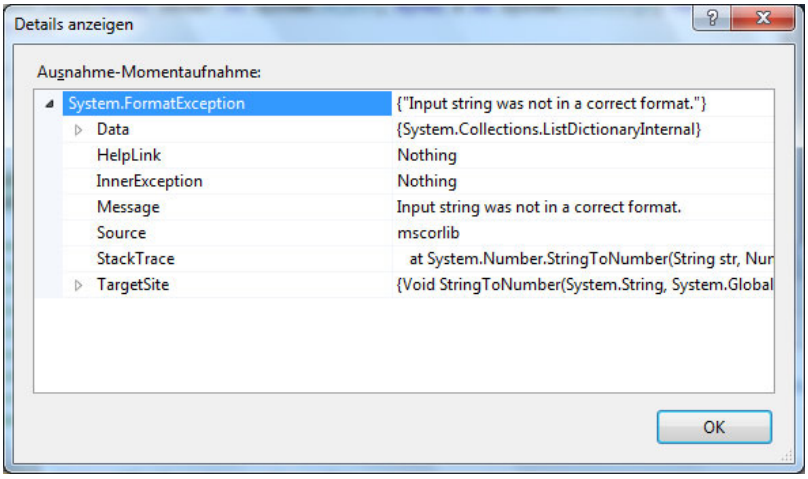
Abbildung 4.1
Troubleshooting-Tipps



Sie erfahren, dass es sich bei dieser Ausnahme um eine `FormatException` handelt, und können somit den entsprechenden `Catch`-Block codieren.

Durch einen Klick auf den Link Details Anzeigen... können Sie sich zusätzliche Informationen zu dieser Ausnahme anzeigen lassen, wie Sie in Abbildung 4.2 sehen können.

Abbildung 4.2
Anzeige der Troubleshooting-Details



Auch innerhalb von Catch-Blöcken können Sie Ausnahmebehandlungen implementieren, um zu vermeiden, dass in Ihrer Ausnahmebehandlung wiederum eine neue Ausnahme auftritt. Der Code, den Sie innerhalb eines Catch-Blocks implementieren, kann also durchaus aufwändiger sein.

Info

Dieses war in Visual Basic 6 nicht möglich, hier führte ein Fehler in der Fehlerbehandlung unweigerlich zum Absturz des Programms.

Genauso ist es möglich, Try-Blöcke ineinander zu schachteln. Die Verschachtelungstiefe ist dabei nicht begrenzt.

4.1.2 Werfen einer Ausnahme

Sie werden vielleicht überrascht sein, dass Sie bei einer Eingabe von 0 als Nenner keine Ausnahme bekommen, sondern das Ergebnis als *+unendlich* oder *-unendlich* – je nachdem, ob der Zähler positiv oder negativ eingegeben wurde – ausgewiesen wird.

Der Grund dafür liegt in der Art und Weise, wie wir dividieren.

```
Convert.ToDouble(txtZaehler.Text)/ _
Convert.ToDouble(txtNenner.Text)
```

An dieser Stelle führen wir eine Fließkommazahlendivision durch (/). Bei dieser Art der Division wird mit einer Annäherung von 0 gerechnet, was somit zu einem Ergebnis von + oder –unendlich führt.

Würden wir eine Integerdivision durchführen (mittels des Operators \), würden wir tatsächlich eine Ausnahme bekommen, nämlich eine DivideByZeroException. Davor müssten wir aber den Nenner und den Zähler der Division vom Typ Long definieren. Ansonsten warnt uns der Compiler (bei eingeschaltetem Option Strict On).

Da wir jedoch ein exaktes Ergebnis ermitteln möchten (bei einer Integerdivision bekommen wir nur Ganzzahlen als Ergebnis zurück), bevorzugen wir die Fließkommazahlendivision. Mit einem unendlichen Wert wollen wir jedoch auch nicht weiterrechnen, also machen wir an dieser Stelle von der Möglichkeit Gebrauch, selbst eine Ausnahme zu werfen.

```
If txtNenner.Text = "0" Then
    Throw New DivideByZeroException()
End If
```

Hier können wir mit dem Schlüsselwort Throw eine eigene Ausnahme auslösen, in diesem Fall eine vom Typ DivideByZeroException.

Innerhalb Ihrer eigenen Klassen wollen Sie vielleicht auch gewisse Fehlerfälle erst einmal behandeln und dann mit Throw an den Client, der Ihre Klasse nutzt, übergeben. Die Fehler, die in Ihren eigenen Klassen geworfen werden, können auch von einem selbst definierten Exception-Typ sein.

Info

Sie werden noch sehen, wie einfach Sie eigene Fehlertypen mittels Vererbung erstellen können.

Dazu werfen Sie mittels `Throw` eine neue Ausnahme von Ihrem eigenen Typ und übergeben den Text und die ursprüngliche Ausnahme an den *Konstruktor* der eigenen Ausnahmeklasse als sogenannte `InnerException`. Diese `InnerException`-Eigenschaft besitzt somit sämtliche Eigenschaften der ursprünglich aufgetretenen Ausnahme.

```
Catch ex As DivideByZeroException
    Throw New MyException(ex.Message, ex)
```

In diesem kleinen Codebeispiel fangen wir die ursprüngliche Exception ab, werfen eine eigene vom Typ `MyException` und übergeben als Ausnahmemeldung die ursprüngliche `Message`, auf die wir über die Objektvariable `ex` zugreifen, sowie das ganze Ausnahmeobjekt `ex` als `InnerException`. Die Eigenschaft `InnerException` ist schreibgeschützt. Sie kann daher nur bei der Instanziierung des Objekts angegeben werden.

Sehr häufig sieht man auch, dass eine Ausnahme abgefangen wird, um den Fehler zu protokollieren. Danach wird die ursprünglich aufgetretene `Exception` wieder weiter geworfen.

Listing 4.2 Weiterwerfen einer Ausnahme

```
Catch ex As DivideByZeroException
    'Protokollierung des Fehlers
    Throw
```

Wie Sie in Listing 4.2 sehen, reicht die bloße Angabe des Schlüsselworts `Throw`, um die ursprüngliche Ausnahme wieder zu werfen.

Achtung

Tatsächlich würde `Throw ex` etwas anderes bewirken als lediglich die Angabe von `Throw`. `Throw ex` löst die ursprüngliche Ausnahme noch mal aus, was zu einer Veränderung des `StackTrace` führt.

Mit dem .NET Framework 2.0 ist die Eigenschaft `Data` der `Exception`-Klasse eingeführt worden. `Data` ist dabei eine **Auflistungsklasse** mit einer **Schlüssel-Wertepaar-Beziehung**. Man kann damit dem `Exception`-Objekt zusätzliche Informationen hinzufügen, die dem Entwickler Hilfestellungen geben können, um zu erkennen, warum eine Ausnahme ausgelöst wurde und wo vielleicht noch Verbesserungspotenzial in der Methode steckt, um solche Ausnahmen im Vorhinein vermeiden zu können. Man kann der `Data`-Eigenschaft zum Beispiel den Wert von Übergabeparametern hinzufügen, um analysieren zu können, wie es zu dieser Ausnahme gekommen ist.

Statt wie oben gerade gezeigt nur eine neue Ausnahme auszulösen und zu werfen, kann man jetzt diesem neuen Ausnahmeobjekt benutzerdefinierte Daten hinzufügen. In diesem Beispiel werden dem neuen `Exception`-Objekt die Werte für den Zähler und den Nenner der `Data`-Collection hinzugefügt (Sie können diese natürlich auch dem ursprünglichen `Exception`-Objekt hinzufügen).

```
Dim myEx As New MyException(ex.Message, ex)
myEx.Data.Add("Zähler", txtZaehler.Text)
myEx.Data.Add("Nenner", txtNenner.Text)
Throw myEx
```

Die `Data`-Eigenschaft kann danach an einer anderen Stelle wieder ausgelesen werden. Jeder Eintrag in der `Data`-Collection ist dabei vom Typ `DictionaryEntry` mit den Eigenschaften `Key` und `Value`. Mittels einer `For Each`-Schleife können jetzt alle Werte der Schlüssel-Wertepaar-Beziehung wieder ausgelesen werden.

4.1.3 Das Schlüsselwort Finally

Nun fehlt uns letztendlich nur noch die Betrachtung des Schlüsselworts `Finally`. `Finally` kann für jeden `Try`-Block einmal auftreten. Seine Verwendung ist optional, es muss also nicht implementiert sein. Der `Finally`-Block wird dabei hinter die `Catch`-Blöcke geschrieben und er wird in jedem Fall ausgeführt.

Sehr oft ist es notwendig, dass bestimmter Code, zumeist das Freigeben von Ressourcen oder andere Aufräumarbeiten, sowohl im positiven Fall wie auch im Fall einer Ausnahme, ausgeführt werden muss. Damit dieser Code nicht nach dem `Try`-Block und innerhalb aller `Catch`-Blöcke mehrfach geschrieben werden muss, wurde der `Finally`-Block eingeführt.

`Finally` wird immer ausgeführt, unabhängig davon, ob eine Ausnahme aufgetreten ist oder nicht, welcher Code in den `Catch`-Blöcken ausgeführt wird, ob die Routine mit `return` verlassen wurde oder ob das Programm innerhalb eines `Catch`-Blocks beendet wird. Insofern unterscheidet sich `Finally` von »gewöhnlichem« Programmcode, der hinter dem `Try`-Konstrukt implementiert ist.

Listing 4.3 soll nun das Ergebnis im `Finally`-Block ausgeben.

Listing 4.3
Strukturierte Fehlerbehandlung mit Finally und When

```
Private mEventLogOn As Boolean
Private Sub btnCalculate_Click( _
    ByVal sender As Object, _
    ByVal e As System.EventArgs) _
    Handles btnCalculate.Click

    Dim ergebnis As Double
    Try
        If txtNenner.Text = "0" Then
                    Throw New FormatException()
        End If
        ergebnis = _
            Convert.ToDouble(txtZaehler.Text) / _
            Convert.ToDouble(txtNenner.Text)
    Catch ex As FormatException
        txtZaehler.Text = "1"
        txtNenner.Text = "1"
        MessageBox.Show(ex.Message & _
            "Geben Sie Zahlen ein, " & _
            "und versuchen Sie es nochmal")
    Catch ex As DivideByZeroException _
```

Listing 4.3 (Forts.) Strukturierte Fehlerbehandlung mit Finally und When

```
        When mEventLogOn = True

        Dim myEx As New MyException( _
            ex.Message, ex)
        myEx.Data.Add("Zähler", txtZaehler.Text)
        myEx.Data.Add("Nenner", txtNenner.Text)
        Throw myEx
    Catch ex As DivideByZeroException _
        When mEventLogOn = False

        txtNenner.Text = "1"
        MessageBox.Show(ex.Message)
    Catch ex As Exception
        MessageBox.Show(ex.Message, _
            "Unerwartete Ausnahme")
    Finally
        MessageBox.Show(ergebnis.ToString())
    End Try
End Sub
```

Außer dem hinzugefügten `Finally`-Block erkennen Sie vielleicht noch einen weiteren Unterschied zu Listing 4.1. Die Definition der Variablen `ergebnis` habe ich vor das Schlüsselwort `Try` geschrieben, da ich auf diese ansonsten im `Finally`-Block nicht zugreifen kann. Diese Variable wäre nämlich ansonsten eine blockweise Variable und hätte ihre Gültigkeit nur im `Try`-Block. Zwar wird die Ausnahmebehandlung erst nach `Finally` abgeschlossen, nämlich mit `End Try`. Nichtsdestotrotz endet der Gültigkeitsbereich des `Try`-Blocks mit dem ersten `Catch`-Block.

Info

Lassen Sie sich nicht verwirren! Das Ende des `Try`-Blocks ist in Visual Basic 10 nicht etwa durch `End Try` gekennzeichnet. Vielmehr schließt `End Try` die Ausnahmebehandlung – also das ganze Konstrukt – ab. Das Ende des inneren `Try`-Blocks wird dagegen durch das erste Auftreten eines `Catch`-Blocks implizit definiert (in der Sprache C# sind die Blöcke bei einer Ausnahmebehandlung durch die geschweiften Klammern deutlich übersichtlicher implementiert).

Durch die Definition vor dem `Try`-Block ist die Variable somit in allen Blöcken sichtbar.

4.1.4 Das Application.ThreadException-Event

Innerhalb von Windows Forms-Anwendungen stellt das `Application`-Objekt mittels `ThreadException` ein Ereignis zur globalen Fehlerbehandlung zur Verfügung. Sie können an diesem Event einen Ereignishandler anbinden, der für alle nicht behandelten Ausnahmen Ihrer Anwendung greift, ohne dass die Anwendung dabei beendet wird oder abstürzt. Sollte eine unbehandelte Ausnahme auftreten, wird der entsprechende Ereignishandler aufgerufen und das Programm läuft anschließend in seinem Kontext weiter.

Der Ereignishandler muss dabei dem Delegate `ThreadException` angefügt werden, bevor die `Run()`-Methode der Applikation aufgerufen wird. Den benötigten Programmcode hierfür fügt man am besten in der `Sub Main`-Routine ein.

In dem `EventArgs`-Parameter `e` vom Typ `System.Threading.ThreadExceptionEventArgs` wird Ihnen dabei das komplette tatsächlich aufgetretene `Exception`-Objekt übergeben. Somit können Sie mittels `e.Exception` auf sämtliche Eigenschaften der aufgetretenen, nicht speziell behandelten Ausnahme zugreifen. Im Parameter `sender` haben Sie Zugriff auf den *Thread*, in dem die Ausnahme aufgetreten ist.

Info

Die Routine `Sub Main` ist in der Regel der Haupteinstiegspunkt in Ihre Windows-Applikationen und wird somit als Erstes durchlaufen. In dieser Routine wird Programmcode beim Start ausgeführt, noch bevor ein Formular angezeigt wird. Im Gegensatz zu C# wird diese Routine dem Visual Basic-Programmierer bei Windows Forms-Anwendungen nicht standardmäßig zur Verfügung gestellt.

Um eine `Main`-Routine anzulegen, fügen Sie Ihrem Projekt zuerst ein neues Modul hinzu. Aus Konventionsgründen sollten Sie diesem neuen Modul den Namen *Program* geben. Fügen Sie dann innerhalb dieses neuen Moduls die `Sub Main` mit dem gewünschten Programmcode hinzu.

```
<STAThread()> Sub Main()
    AddHandler Application.ThreadException, AddressOf GlobalExceptionHandler
    Application.Run(New Form1())
End Sub
Public Sub GlobalExceptionHandler(ByVal sender As Object, _
ByVal e As System.Threading.ThreadExceptionEventArgs)
    MessageBox.Show(e.Exception.Message)
End Sub
```

Abbildung 4.3
Sub Main im Modul Program.vb

Anschließend müssen Sie in den Projekteigenschaften noch einstellen, dass Ihr *Startup-Objekt* die `Sub Main` und nicht irgendein Formular ist.

Wählen Sie dazu aus dem Kontextmenü für den Eintrag *MyProject* im Projektmappen-Explorer den Eintrag Öffnen. Im anschließenden Dialog muss das Register Anwendung ausgewählt sein. In der Auswahlliste Startupobjekt muss dann der Eintrag Sub Main ausgewählt werden. Dieser Eintrag ist jedoch in der Liste erst enthalten, wenn das Kontrollkästchen Anwendungsframework aktivieren deaktiviert wird.

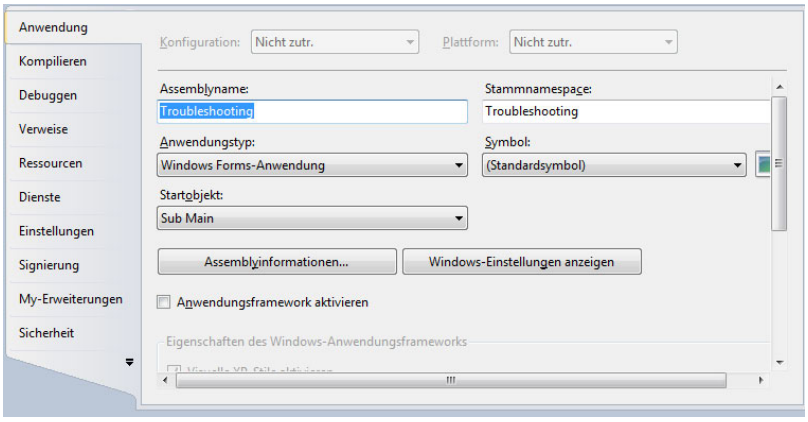


Abbildung 4.4
Projekteigenschaften zur Auswahl des Startobjekts

Wenn Sie die globale Fehlerbehandlung nun innerhalb der Entwicklungsumgebung testen, wird weiterhin das Programm an der Position, an dem die Ausnahme auftritt, angehalten und die Entwicklungsumgebung verhält sich wie bei jeder anderen nicht behandelten Ausnahme. Wenn Sie die Exe-Datei jedoch direkt starten oder die Anwendung ohne Debugger starten (Strg + F5), werden Sie feststellen, dass der Ereignishandler greift und Ihr Programm in einem definierten Zustand weiter korrekt ausgeführt wird.

4.2 Unstrukturierte Ausnahmebehandlung

Aus abwärtskompatiblen Gründen steht uns in Visual Basic 10, aber auch nur in dieser .NET-Sprache, die unstrukturierte Ausnahmebehandlung zur Verfügung, so wie Sie diese vielleicht aus einer Visual Basic-Vorgängerversion kennen.

Info

Die unstrukturierte Fehlerbehandlung war in der allerersten Beta-Version von Visual Basic .NET nicht enthalten. Nach einem Aufschrei der VB-Entwickler entschied sich Microsoft jedoch, diese Art der Ausnahmebehandlung in VB.NET mit aufzunehmen.

Bei der unstrukturierten Ausnahmebehandlung leiten Sie einen ausnahmebehandelten Codeblock mit der `On Error`-Anweisung ein.

Dabei gibt es verschiedene Optionen, was bei einem Fehler (`On Error`) passieren kann, die ich hier kurz beschreiben möchte:

- On Error GoTo Line

  Mit dieser Ausnahmebehandlung wird beim Auftreten einer Ausnahme an den hinter `GoTo` definierten Haltepunkt gesprungen. Dieser Haltepunkt muss in derselben Methode definiert sein, in der auch die `On Error GoTo`-Anweisung steht.

  Hier können Sie mit `Resume` die fehlerhafte Zeile wiederholen, mit `Resume Next` die Anweisung ausführen, die nach der fehlerhaften Zeile kommt, das Programm fortsetzen oder die Routine verlassen.

  Vor der Sprungmarke sollte der Code mittels `Exit Sub` oder `Return` verlassen werden. Würde dieser Code ausgeführt werden, wenn keine Ausnahme vorliegt, führt der Aufruf einer `Resume`-Anweisung zu einer Ausnahme, die nicht mehr behandelt werden kann.

  Wird innerhalb der ursprünglichen Methode eine weitere Methode ohne Ausnahmebehandlung aufgerufen, so wird bei Auftreten einer Ausnahme an die Sprungmarke der aufrufenden Methode gesprungen.

- On Error Resume Next

  Mit dieser Anweisung wird jede Ausnahme ignoriert und einfach die nächste Anweisung ausgeführt.

  Wird innerhalb der ursprünglichen Methode eine weitere Methode ohne Ausnahmebehandlung aufgerufen, so wird in der aufrufenden Methode die `Resume Next`-Anweisung ignoriert. Sie sollten in der aufgerufenen Methode auch eine Ausnahmebehandlung implementieren.

- On Error GoTo 0

  Hebt jegliche vorangegangene Ausnahmebehandlung innerhalb einer Methode auf.

- On Error GoTo -1

  Hebt jede Ausnahmebehandlung innerhalb einer Methode auf.

Bei dieser Art der Ausnahmebehandlung steht Ihnen auch kein Objekt vom Typ `Exception` zur Verfügung, sondern nur ein zu anderen .NET-Sprachen nicht kompatibles `err`-Objekt mit den beiden wichtigen Eigenschaften `err.Number` und `err.Description`. Eine spezifische Ausnahmebehandlung ist an dieser Stelle nur durch ein Unterscheiden der Fehlernummer (`err.Number`) in einem `Select Case`-Block möglich.

Bitte verwenden Sie diese Art der Ausnahmebehandlung auch tatsächlich nur bei einer Programmmigration. Bei neu entwickeltem Code verwenden Sie ausschließlich eine strukturierte Ausnahmebehandlung. Microsoft selbst rät dringend von dieser Art ab, da sie die Performance eines Programms verschlechtern kann und schwer zu debuggenden und wartbaren Code erzeugt.

5

Visual Basic 10 OOP

Im Laufe seiner Geschichte entwickelte sich Visual Basic von einer reinen strukturellen Interpretersprache immer mehr zu einer Compiler-Sprache mit objektorientierten Ansätzen. Ab der Version 5 war es bereits möglich, mit Klassen zu arbeiten, doch fehlte noch eines der wichtigsten objektorientierten Features, nämlich die Vererbung. Außerdem konnte man in dieser Version, genauso wie auch in der Nachfolgeversion VB6, weiterhin strukturell programmieren. Erst mit der Einführung von .NET wurde aus VB eine durchgängig objektorientierte Sprache, deren Features in diesem Kapitel erläutert werden.

Doch kommen wir nun erst einmal zu den Grundsätzlichkeiten bei der Erstellung einer Klasse.

5.1 Klassen

Eine Klasse beschreibt die Art und Weise, wie Objekte vom Typ dieser Klasse erstellt werden. Die Definition einer Klasse ist somit eine Schablone für Objekte dieses Typs. Innerhalb einer Klasse wird definiert, welche Eigenschaften (die Daten eines Objekts) und welche Methoden (die Logik eines Objekts) für ein Objekt zur Verfügung stehen. Für die Kommunikation mit anderen Objekten werden Events (Ereignisse) implementiert.

Durch die Instanziierung wird ein Objekt, eine Instanz der Klasse, erzeugt. Dabei wird der sogenannte Konstruktor der Klasse aufgerufen, mit dem Eigenschaftswerte des Objekts vorbelegt werden können.

Bei der Definition einer Klasse definieren Sie somit:

- einen oder mehrere Konstruktoren
- Member-Variablen
- Eigenschaften/Properties zum Zugriff auf die Member-Variablen
- Methoden
- Events

Die Klassendefinition wird dabei durch die Schlüsselwörter `Class` und `End Class` begrenzt.

```
Class Klassenname
    'Logik der Klasse
End Class
```

Eine Visual Basic-Klasse wird innerhalb einer Visual Basic-Codedatei (*.*vb*) definiert. In einer solchen *vb*-Datei können eine oder auch mehrere Klassendefinitionen geschrieben werden, auch wenn Letzteres für die Übersichtlichkeit in Ihrem Projekt nicht förderlich ist.

5.1.1 Eigenschaften

Die Daten eines Objekts werden durch Eigenschaften (**Properties**) implementiert. Die Werte dieser Daten werden dabei in Member-Variablen gespeichert, die auf Klassenebene definiert werden, aber von außen nicht zugreifbar sind. Für den Zugriff auf diese Daten von außerhalb der Klasse werden deswegen Properties definiert, die einen genormten Lese- und Schreibzugriff auf diese Daten definieren.

Info

Die Daten eines Objekts werden in anderen Programmiersprachen sehr oft als Attribute bezeichnet. Attribute besitzen aber im .NET Framework eine gänzlich andere Bedeutung (siehe Kapitel 5 Attribute).

Der Vorteil von Properties gegenüber öffentlichen (`public`) Variablen besteht darin, dass es eine ganz zentrale Stelle innerhalb der Applikation gibt, an der auf ein bestimmtes Datum der Klasse zugegriffen wird. Somit kann man an einer einzigen Stelle Plausibilitätsüberprüfungen beim Setzen eines Werts durchführen oder auch bei Änderung von Werten bestimmte Events ausführen. Dies wäre bei öffentlichen Variablen nicht möglich, ganz im Gegenteil müssten die dementsprechenden Überprüfungen an sehr vielen unterschiedlichen Stellen der Applikation aufgerufen werden, was den Programmcode sicherlich nicht wartbar und übersichtlich macht.

In Listing 5.1 sehen Sie ein Beispiel für eine Property mit dem Namen »PersonalNummer« mit zugehöriger Definition der Member-Variablen.

Listing 5.1
Beispiel für eine Property

```
Private mPersonalNummer As String
Public Property PersonalNummer() As String
    Get
        Return mPersonalNummer
    End Get
    Set(ByVal value As String)
        mPersonalNummer = value
    End Set
End Property
```

Der `Get`-Teil gibt dabei die Variable nach außen bekannt, während der Wert für diese Eigenschaft im `Set`-Zweig mittels der Variablen `value` gesetzt wird.

Diese Lese-/Schreibroutinen werden dabei durch das Setzen oder Auslesen eines Werts aufgerufen.

Die folgende Codezeile ruft dabei innerhalb der Klasse den `Get`-Zweig der Property auf und gibt den Wert in einem Meldungsfenster aus:

```
MessageBox.Show(MyKlasseVariable.PersonalNummer)
```

während die folgende Zeile den `Set`-Zweig aufruft, wobei der Wert auf der rechten Seite der Zuweisung der Variablen `Value` innerhalb des `Set`-Zweigs zugewiesen wird.

```
MyKlasseVariable.PersonalNummer = "2606-C80-162"
```

Plausibilitätsüberprüfungen

Es empfiehlt sich, beim Setzen des Werts (also innerhalb des `Set`-Zweigs einer Eigenschaft) auch eine Plausibilitätsprüfung auf den übergebenen Wert durchzuführen. Der Visual Basic-Compiler wird verhindern, dass an diese Eigenschaft ein falscher Datentyp übergeben wird, jedoch kann auch ein passender Datentyp logisch falsch sein.

Nehmen wir zum Beispiel eine Eigenschaft `Alter` (in Jahren) einer Person, die vom Datentyp als `Short` definiert ist. Jegliche negative Zahl ist vom Datentyp her zwar korrekt, jedoch wird es keine Person mit einem negativen Alter geben. Auch ein Alter jenseits der 150 halte ich für eine sehr gewagte Zuweisung (zumindest in den meisten Applikationen).

Nun stellt sich die Frage, wie man auf solch logisch falsche Zuweisungen reagiert. In Schulungen höre ich dabei sehr oft die Antwort: »Man gibt einfach eine Messagebox aus.« Gute Idee, aber leider nicht die richtige. Zum einen würde diese Anweisung innerhalb einer Klassendefinition stehen und aus dieser heraus eine Kommunikation mit dem Benutzer darstellen. Der einzige, der jedoch für die Kommunikation mit dem Benutzer zuständig ist, ist der Entwickler der Benutzeroberfläche. Wir würden diesen mit so einer Anweisung überrumpeln, denn er könnte nicht verhindern, dass diese Meldung ausgegeben wird. Und weder wissen wir, ob er an dieser Stelle überhaupt eine Meldung ausgeben will, noch wissen wir, ob ihm unser Ausgabetext (denken Sie zum Beispiel nur an Mehrsprachigkeit) gefallen würde.

Zum anderen wissen wir auch nicht, in welcher Art von Applikation unsere Klasse verwendet wird. In einer serverseitig programmierten Webapplikation (wie zum Beispiel ASP.NET) würde zum Beispiel diese Meldung auf dem Webserver erscheinen und nicht auf dem Client, der die fehlerhafte Eingabe verursachte.

Solche Ausgaben würden definitiv die Wiederverwendbarkeit unserer Klasse extrem einschränken. Außerdem würde der Programmcode nach der Ausgabe weiterlaufen, ohne dass auf die logisch falsche Zuweisung reagiert wird.

Auch das Aufrufen eines Ereignisses ist hier nur die halbe Wahrheit, denn auf Ereignisse muss man nicht zwingend reagieren.

Die einzige Möglichkeit, den Entwickler der GUI dazu zu zwingen, auf die logisch falsche Zuweisung zu reagieren, besteht darin, tatsächlich mit dem Schlüsselwort `Throw` eine `Exception` auszulösen, auf die dann im Client reagiert wird.

Dies wird in Listing 5.2 dargestellt:

Listing 5.2 Erweiterte Property

```
Private mAlter As Short
Public Property Alter() As Short
    Get
        Return mAlter
    End Get
    Set(ByVal value As Short)
        If value >= 0 And value < 150 Then
            mAlter = value
        Else
            Throw New Exception ("Ungültiges Alter")
        End If
    End Set
End Property
```

ReadOnly-Eigenschaften

Sehr oft wird innerhalb einer Klasse die Anforderung bestehen, bestimmte Eigenschaften als schreibgeschützte Properties zu implementieren. Das bedeutet, dass der Wert zwar ausgelesen, jedoch nur über Funktionalität verändert werden kann und nicht über eine einfache Wertzuweisung (auch wenn ich mir oft wünschte, ich könnte meinen Kontostand am Geldautomaten einfach setzen).

Für solche Zwecke gibt es die Möglichkeit, mit dem Schlüsselwort `ReadOnly` schreibgeschützte Eigenschaften zu implementieren. Der `Set`-Zweig muss hierbei logischerweise weggelassen werden.

Listing 5.3 Schreibgeschützte Property

```
Private mGehalt As Double
Public ReadOnly Property Gehalt() As Double
    Get
        Return mGehalt
    End Get
End Property
```

WriteOnly-Eigenschaften

Das Gegenteil zu schreibgeschützten Eigenschaften sind lesegeschützte Eigenschaften. Hier kann der Wert zwar gesetzt, jedoch nicht mehr ausgelesen werden. Ein Anwendungsbeispiel hierzu sind PINs oder sonstige Arten von Passwörtern.

Mit dem Schlüsselwort `WriteOnly` können Sie lesegeschützte Properties definieren, bei denen dann der `Get`-Zweig nicht implementiert wird.

Listing 5.4 Lesegeschützte Property

```
Private mPasswort As String
Public WriteOnly Property Passwort() As String
    Set(ByVal value As String)
        mPasswort = value
    End Set
End Property
```

Default-Eigenschaften

Standardeigenschaften wurden unter VB 6 sehr häufig verwendet, jedoch führte das nicht unbedingt zu sehr leserlichem Programmcode. Deswegen wurde die Verwendung von Standardeigenschaften unter NET sehr eingeschränkt. Die Aussage, es gäbe unter .NET keine Standardeigenschaften mehr, ist jedoch falsch, sie können eben nur noch eingeschränkt verwendet werden.

Die Voraussetzung, um eine Property mit dem Schlüsselwort `Default` als Standardeigenschaft zu definieren, ist, dass es sich dabei um eine Auflistung von Werten handelt, sprich um eine Auflistung, auf die über einen Index oder einen Schlüsselbegriff zugegriffen wird. Im .NET-Sprachgebrauch würde man eine Standardeigenschaft mit dem Namen `Item` bezeichnen, dies ist jedoch keine Pflicht.

```
Private mItem(10) As String
Default Public Property Item
  (ByVal Index As Integer) As String
    Get
        Return mItem(Index)
    End Get
    Set(ByVal value As String)
        mItem(Index) = value
    End Set
End Property
```

Diese Property kann danach später in folgender Form aufgerufen werden:

```
MyKlasse(i)
```

oder auch mit der Angabe des Eigenschaftsnamens:

```
MyKlasse.Item(i)
```

Unterschiedliche Gültigkeitsbereiche im Get- und Set-Zweig

Trotz `ReadOnly`- und `WriteOnly`-Eigenschaften will man manchmal `Get`- und `Set`-Zweige mit unterschiedlichen Gültigkeitsbereichen definieren.

Info

Dies war in VB 6 bereits möglich, in den ersten Versionen von VB.NET jedoch nicht. Erst ab der Version Visual Basic 2005 wurde diese Möglichkeit von Microsoft wieder implementiert.

Das macht vor allem dann Sinn, wenn man beim Ändern von bestimmten Werten Events an ganz zentralen Stellen aufrufen oder auch ganz trivial Plausibilitätsüberprüfungen durchführen will, selbst wenn die Variable aus der eigenen Klasse heraus gesetzt wird.

Um dies zu bewerkstelligen, kann man einfach vor dem entsprechenden `Get`- oder `Set`-Zweig einen unterschiedlichen Gültigkeitsbereich definieren.

Im folgenden Codebeispiel kann man nur innerhalb der Klasse auf den `Set`-Zweig zum Setzen des Alters zugreifen.

Listing 5.5
Property mit unterschiedlichen Gültigkeitsbereichen

```
Private mAlter As Short
Public Property Alter() As Short
    Get
        Return mAlter
    End Get
    Private Set(ByVal value As Short)
        If value >= 0 And value < 150 Then
            mAlter = value
        Else
            Throw New Exception ("Ungültiges Alter")
        End If
    End Set
End Property
```

Wird bei einer Property kein Gültigkeitsbereich angegeben, so ist sie standardmäßig `Public`.

Automatisch implementierte Eigenschaften

Mit Visual Basic 10 wurden sogenannte automatisch implementierte Eigenschaften eingeführt.

Die Syntax für eine Property kann nun auch stark verkürzt wie folgt geschrieben werden:

```
Public Property Name As String
```

Wie Sie sehen, müssen Sie nicht mehr die private Variable und den *Get*- und *Set*-Zweig implementieren, das macht der Compiler für Sie. Sie müssen lediglich darauf achten, dass Sie in der Klasse nicht zusätzlich eine Variable *_Name* definieren, denn so heißt die korrespondierende Member-Variable, die der Compiler für die Eigenschaft automatisch im Hintergrund anlegt.

Die beschriebene Kurzform können Sie jedoch nur verwenden, wenn Sie keine Plausibilitätsprüfungen implementieren.

5.1.2 Methoden

Die Logik oder Funktionalität eines Objekts wird durch Methoden innerhalb der Klasse implementiert. Eine Methode kann dabei eine öffentliche Funktion oder eine Prozedur (Sub) sein.

Info

Eine *Sub* entspricht einer Funktion ohne Rückgabewert. In anderen Programmiersprachen wird das als *void*-Function bezeichnet.

Wenn Sie bei einer Methode keinen **Modifier** angeben, ist diese standardmäßig `Public`.

An eine Methode können Sie keinen, einen oder beliebig viele Parameter übergeben. Parameter werden in Visual Basic 10 auch beim Aufruf grundsätzlich in Klammern geschrieben.

Info

In VB 6 gab es für die Notation beim Aufruf einer Methode viele Regeln mit noch mehr Ausnahmen, wann und ob die Parameter in Klammern geschrieben werden müssen oder nicht.

Die Parameterübergabe erfolgt in den neuen Visual Basic-Versionen standardmäßig `ByVal` und nicht wie in der Vergangenheit `ByRef`. Der Unterschied zwischen beiden Varianten der Parameterübergabe liegt darin, dass einmal der Wert der Variablen übergeben wird (*ByVal*) und zum anderen eine Referenz auf die Variable (*ByRef*). Im Standardfall `ByVal` wird somit eine Kopie der Variablen übergeben, was als Konsequenz bedeutet, dass die aufgerufene Methode den Wert der Originalvariablen nicht verändern kann. Im Fall von `ByRef` wird hingegen die Speicheradresse der Variablen übergeben, was der aufrufenden Methode die Möglichkeit gibt, direkt in diese Speicheradresse zu schreiben und somit den Originalwert zu manipulieren.

Wenn Sie innerhalb einer Klasse auf eine Property zugreifen wollen, können Sie optional das Schlüsselwort `Me` vor den Property-Namen setzen, um zu verdeutlichen, dass es sich dabei um eine Eigenschaft dieser Instanz (`Me`) der Klasse handelt.

In Listing 5.6 wird eine `Sub` definiert, über die das Alter einer Person gesetzt werden kann.

Listing 5.6
Definition einer Sub

```
Public Sub SetzeAlter(ByVal neuesAlter As Short)
    Me.Alter = neuesAlter
End Sub
```

Während im nächsten Beispiel über eine Funktion das Alter zu einem bestimmten Zeitpunkt als Rückgabewert zurückgegeben wird:

Listing 5.7
Definition einer Funktion

```
Public Function AlterAm
  (ByVal Datum As DateTime) As Short
    Dim DiffJahre As Short = System.DateTime.Now.Subtract
       (Datum).TotalDays / 365
    Return Me.Alter - DiffJahre
End Function
```

In beiden Beispielen wird der Parameter `ByVal` übergeben. Wollten Sie den Parameter nicht als Wert übergeben, sondern als Referenz, so muss einfach `ByRef` vor den entsprechenden Parameter gesetzt werden.

5.1.3 Overloading

Mittels **Overloading** hat man die Möglichkeit, innerhalb einer Klasse mehrere gleichnamige Methoden, Properties und Konstruktoren zu definieren, die sich jedoch durch die Parametersignatur eindeutig unterscheiden lassen müssen. Die Signatur einer/eines Methode/Property/Konstruktors setzt sich dabei aus der Anzahl und den Datentypen der Parameter zusammen. Durch den Aufruf erkennt der Compiler aufgrund der übergebenen Parameter, welche Variante der überladenen Methode ausgeführt werden soll. Das Ändern

eines Parameternamens mit identischem Datentyp ist keine Änderung der Signatur, ebenso wenig das Ändern der Art und Weise der Parameterübergabe (`ByVal` oder `ByRef`).

Sie haben die Möglichkeit, sowohl Properties wie auch Methoden zu überladen, das Überschreiben einer Property mit einer Methode (Function oder Sub) ist jedoch nicht möglich. Möglich ist es jedoch, eine Funktion mit einer Prozedur (ohne Rückgabewert) zu überladen. Überladene Funktionen können auch unterschiedliche Rückgabewerte besitzen, das alleinige Ändern des Rückgabewerts ist dabei jedoch keine Änderung der Signatur (und somit ist das Überladen ohne Änderung eines Parameters nicht zulässig!).

Um ein überschriebenes Member als überladen zu kennzeichnen, kann man diese Member optional mit dem Schlüsselwort `Overloads` versehen. Wenn `Overloads` bei einer Version eines überladenen Members verwendet wird, dann muss es auch bei allen weiteren Versionen dieses Members verwendet werden. Das Verwenden des Schlüsselworts `Overloads` ist jedoch bei überladenen **Konstruktoren** (mehr dazu in Abschnitt 7 Konstruktoren und Destruktoren) nicht gestattet.

Beispiel für eine überladene Methode mit unterschiedlicher Anzahl von Parametern und Datentypen:

Listing 5.8 Überladene Methode

```
'Variante 1 mit zwei Integervariablen
Public Overloads Function BerechneWas(ByVal x As Integer,
    ByVal y As Integer) As Integer

    Return x + y
End Function
'Variante 2 mit drei Integervariablen
Public Overloads Function BerechneWas(ByVal x As Integer,
    ByVal y As Integer, ByVal z As Integer) As Integer

    Return x + y + z
End Function
'Variante 3 mit zwei Long-Variablen
Public Overloads Function BerechneWas(ByVal x As Long,
    ByVal y As Long) As Long

    Return x + y
End Function
'Variante 4 mit drei Long-Variablen
Public Overloads Function BerechneWas(ByVal x As Long,
    ByVal y As Long, ByVal z As Long) As Long

    Return x + y + z
End Function
```

Im obigen Beispiel ist die Methode `BerechneWas` vier Mal mit unterschiedlicher Parametersignatur definiert. Würden in der Variante drei und vier die Parameter wiederum vom Typ `Integer` sein, der Rückgabewert jedoch vom Typ `Long`, wäre die Überladung nicht zulässig, da alleine die Änderung des Rückgabewerts keine Änderung der Signatur darstellt.

Optionale Parameter

Eine Alternative zur Überladung sind die bereits aus den Vorgängerversionen von Visual Basic bekannten **optionalen** Parameter. Diese sind weiterhin in Visual Basic 10 verfügbar, jedoch nicht in allen .NET-Sprachen. (C# unterstützte zum Beispiel lange Zeit keine optionalen Parameter, erst in der aktuellen Version wurden diese auch in C# eingeführt.)

Sie sollten grundsätzlich überladene Methoden den optionalen Parametern vorziehen. Allerdings können Sie für optionale Parameter Standardwerte setzen.

Achtung

Im Gegensatz zu VB 6 müssen optionale Parameter in Visual Basic 10 einen Standardwert zugewiesen bekommen, dieser kann aber jetzt auch *Nothing* sein.

```
Public Function BerechneWasOptional
  (ByVal x As Integer, ByVal y As Integer,
   Optional ByVal z As Integer = 0) As Integer

    Return x + y + z
End Function
```

Optionale Parameter können auch in überladenen Methoden eingesetzt werden, sie müssen aber grundsätzlich am Ende der Parameterliste stehen.

Gerade wenn Sie viel Office-Automation (Steuerung von Excel oder Word) programmieren, werden Sie sehr häufig optionale Parameter in den Office-Funktionalitäten vorfinden.

ParamArrays

Sollte an eine Methode eine nicht überschaubare Anzahl von Parametern übergeben werden können, wäre die Definition von x-Varianten einer Methode sehr umfangreich und aufwändig. Aus diesem Grund können Sie auch ein Parameter-Array als Übergabeparameter definieren. Es können dann für den als `ParamArray` definierten Parameter kein Wert, ein Wert oder beliebig viele Werte übergeben werden.

Diese Form von Überladung sollte eingesetzt werden, wenn jeder übergebene Wert in dieselbe Logik einfließt und die Anzahl der übergebenen Werte nicht von vornherein feststeht.

Folgende Definition

```
Public Function BerechneWasParamArray
    (ByVal ParamArray x() As Integer) As Integer
```

entspricht dabei der Definition der überladenen Methoden

```
Public Function BerechneWas() As Integer
```

oder

```
Public Function BerechneWas
  (ByVal x1 As Integer) As Integer
```

oder

```
Public Function BerechneWas
  (ByVal x1 As Integer,
   ByVal x2 As Integer) As Integer
```

oder

```
Public Function BerechneWas
  (ByVal x1 As Integer, ByVal x2 As Integer,
   ByVal x3 As Integer) As Integer
```

und so weiter.

Flexibler, aber auch gefährlicher wäre die Option, das Parameter-Array ohne Datentypen zu definieren. Somit gäbe es die Möglichkeit, Parameter mit unterschiedlichen Datentypen zu übergeben. Das funktioniert jedoch nur, wenn die strikte Typenüberprüfung mit **Option Strict Off** deaktiviert ist. Der Compiler könnte nicht mehr überprüfen, welche Daten übergeben werden und ob unter diesen Daten nicht unterstützte Datentypen vorhanden sind. Dies würde einen sehr hohen Testaufwand bedeuten und der Compiler würde einen weniger performanten Code erstellen. Auch wenn es diese Möglichkeit gibt, sollte man Abstand davon nehmen, sie zu nutzen.

5.1.4 Ereignisse

Mittels Ereignissen, oder auch Events, ist es möglich, mit anderen Objekten zu kommunizieren. Dabei wird durch ein Ereignis eine Nachricht an ein anderes Objekt geschickt, das darauf reagieren kann oder auch nicht. Die Nachricht wird dabei nur in eine Richtung gesendet, eine Rückantwort erfolgt nicht. Dem aufrufenden Objekt wird auch nicht mitgeteilt, ob jemand auf das Ereignis reagiert hat. Wie das empfangende Objekt auf ein Ereignis reagiert, wird im auslösenden Objekt in keiner Weise beschrieben. In der Klasse wird lediglich der Event definiert und an einer bestimmten Stelle ausgelöst.

Einem Event können bestimmte Parameter mit Informationen übergeben werden. Diese Parameter sind bei der Definition anzugeben und beim Auslösen des Events mit Werten zu befüllen. Unter .NET gibt es für diese Parameter eine Standardsyntax, die durch einen Standardeventhandler definiert ist. Bei sämtlichen Events in Windows Forms oder auch in WebForms werden jeweils zwei Parameter übergeben: ein Parameter vom Typ `Object`, der angibt, wer den Event ausgelöst hat, und ein Parameter vom Typ `EventArgs` oder einem von dieser Klasse abgeleiteten Typ. Somit hat man eine einheitliche typenkompatible Event-Syntax für alle Ereignisse.

An der Art und Weise, wie Ereignisse innerhalb einer Klasse definiert und ausgelöst werden, hat sich gegenüber dem alten Visual Basic 6 nicht viel geändert, jedoch ist das Konzept, das hinter den Events steckt, vollkommen anders als in der Vorgängerversion. Events unter .NET basieren auf sogenannten Delegates und Ereignisroutinen, die auf gleiche Events reagieren, sie benötigen deshalb eine einheitliche Signatur. Ein Delegate ist eine Art typensicherer (wegen einheitlicher Signatur) Zeiger auf die Adresse einer

Funktion (die Adresse der Ereignisroutine). Ein Delegate kann auch ein Array von Funktionsadressen enthalten, was bedeutet, dass bei einem Auslösen des Events mehrere Ereignisroutinen gleichzeitig aufgerufen werden. Voraussetzung ist nur, dass sämtliche Funktionen innerhalb dieses Arrays eine einheitliche Signatur besitzen.

Durch dieses neuartige Delegate-Konzept hat man jedoch bei weitem mehr Möglichkeiten, Routinen zu definieren, die auf Events reagieren. So kann man jetzt dynamisch Funktionsadressen einem Event hinzufügen und auch wieder wegnehmen oder gleichzeitig eine Funktion verschiedenen Events zuordnen, ohne innerhalb verschiedener Ereignisroutinen einen Aufruf an die entsprechende Funktion zu implementieren.

Info

Delegates sind eines der interessantesten Konzepte, die mit .NET eingeführt wurden. Mehr zu Delegates erfahren Sie in Abschnitt 5.6.

Definition und Auslösen eines Ereignisses

Kommen wir aber nun erst einmal dazu, wie ein Ereignis definiert und ausgelöst wird. Die Definition eines Ereignisses geschieht mit folgender Zeile innerhalb einer Klasse:

```
Public Event AlterChanged()
```

Oder besser in der .NET-Event-Syntax:

```
Public Event AlterChanged
   (ByVal sender As Object, ByVal e As EventArgs)
```

Um ein Event auszulösen, steht die Anweisung `RaiseEvent` zur Verfügung. In unserem kleinen Beispiel könnten wir nun an der Stelle, wo das Alter einer Person geändert wird, das entsprechende Ereignis auslösen.

Listing 5.9
Property mit Event-Aufruf

```
Public Property Alter() As Short
    Get
        Return mAlter
    End Get
    Private Set(ByVal value As Short)
        If value >= 0 And value < 150 Then
            mAlter = value
            RaiseEvent AlterChanged(Me, New EventArgs)
        Else
            Throw New Exception("Ungültiges Alter")
        End If
    End Set
End Property
```

Statt der Standardklasse `EventArgs` kann man natürlich selbst eine eigene Klasse definieren (durch Vererbung ist das auch sehr einfach), in der man eigene zusätzliche Properties definieren kann. Das zugrunde liegende Delegate wird dabei in Visual Basic 10 selbst erzeugt, ohne dass man sich darum zu kümmern hat.

Auf Events reagieren

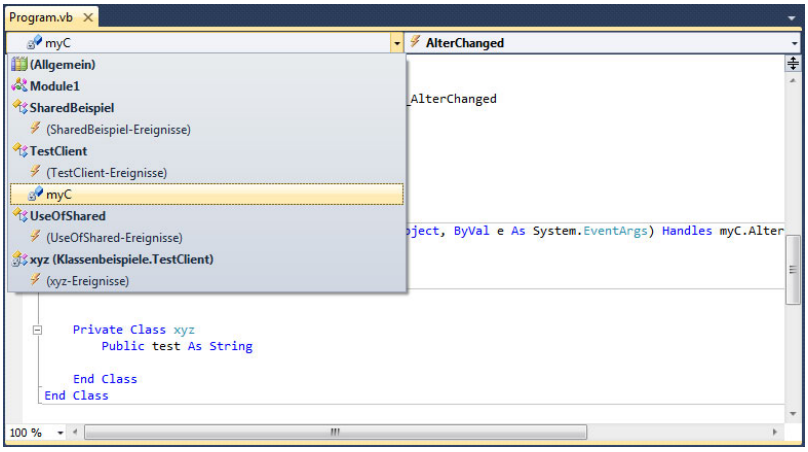
Um in einem beliebigen Client nun auf den entsprechenden Event zu reagieren, hat man, dank des Delegate-Konzepts, verschiedene Möglichkeiten.

Zum einen kann man mit dem Schlüsselwort `WithEvents` eine klassenweite Objektvariable definieren, mit der man auf Ereignisse eines bestimmten Typs reagieren kann. Mit der Anweisung `Handles` am Ende eines Prozeduraufrufs kann man diese Prozedur dann für dieses Ereignis registrieren.

Dies ist die typische Vorgehensweise, wie es zum Beispiel auch in Windows Forms realisiert ist.

Nachdem in einem Client ein Objekt mit dem Schlüsselwort `WithEvents` definiert wurde, kann man aus den beiden Combo-Boxen im Code-Editor die dementsprechende Ereignisroutine auswählen.

Abbildung 5.1
Event-Auswahl



Wie die Prozedur in diesem Fall heißt, ist völlig egal, wichtig ist lediglich die Angabe hinter dem `Handles`-Schlüsselwort. Dabei wird mittels der Syntax `Objektvariable.Eventname` angegeben, dass diese Prozedur ausgeführt wird, wenn für die Objektinstanz das dementsprechende Ereignis ausgelöst wird.

Sehr praktisch ist dieser Weg, wenn eine Prozedur auf unterschiedliche Events reagieren soll, denn hinter der `Handles`-Anweisung können mit Kommata getrennt mehrere Events angegeben werden, auf die reagiert werden soll.

In Windows Forms ist diese Syntax sehr häufig anzutreffen. Innerhalb einer Form kann man ein und dieselbe Funktionalität oft auf unterschiedlichem Weg aufrufen (z.B. Button, Menü, Kontextmenü, Symbolleiste). Anstatt für jeden Aufruf eine eigene Prozedur anzulegen, genügt es nun, eine einzige Prozedur zu definieren und sämtliche Möglichkeiten über die Handles-Anweisung zu verbinden.

Eine andere Möglichkeit ist die dynamische Zuweisung von Events zu bestimmten Prozeduren. Dies ist vor allem dann interessant, wenn man

Oberflächen dynamisch aufbauen möchte und die Steuerelemente per Programmcode erzeugt, oder auch dann, wenn man Objekte nicht klassenweit definieren möchte.

Eine dynamische Zuweisung von Events erfolgt mittels der Anweisung `AddHandler` beziehungsweise die Aufhebung dieser Zuweisung mit der Anweisung `RemoveHandler`.

Die Syntax hierzu sieht wie folgt aus:

```
AddHandler myC.AlterChanged, AddressOf myC_AlterChanged
```

beziehungsweise:

```
RemoveHandler myC.AlterChanged, AddressOf myC_AlterChanged
```

Dabei wird dem Handler für den Event einer Objektinstanz (in diesem Fall der Event `AlterChanged` des Objekts `myc`) die Adresse einer Funktion (welche die identische Signatur wie der Event besitzen muss) hinzugefügt beziehungsweise wieder entfernt.

Durch diese programmgesteuerte Zuweisung von Ereignisroutinen bieten sich dem Entwickler jetzt ganz neue Möglichkeiten. Haben Sie sich noch nie gewünscht, dass bestimmte Ereignisse erst abgefeuert werden, wenn das Formular komplett geladen und initialisiert ist, oder dass ein Event nur ein einziges Mal ausgeführt wird. Mit `AddHandler` und `RemoveHandler` haben Sie jetzt die Möglichkeiten, dies alles zu bewerkstelligen.

5.1.5 Gültigkeitsbereiche

Gültigkeitsbereiche von Klassen, Methoden, Eigenschaften, Klassen-Members, Events, Delegates, Interfaces, Variablen werden durch **Access Modifier** bei deren Deklaration angegeben.

In der Tabelle 5.1 sehen Sie eine Übersicht über die in Visual Basic 10 verfügbaren Zugriffsmodifier und deren Bedeutung.

Tabelle 5.1 Übersicht der Zugriffsmodifier

AccessModifier	Bedeutung
`Public`	`Public` bedeutet, dass das so definierte Element an jeder Stelle des Projekts, in dem es definiert ist, und von jedem Projekt, das auf dieses Projekt referenziert, zugreifbar ist. Die einzige Ausnahme besteht darin, wenn der übergeordnete Container einen eingeschränkteren Gültigkeitsbereich besitzt. Es ist also nicht möglich, von jeder Stelle des Codes auf eine `Public`-Methode zuzugreifen, die innerhalb einer privaten Klasse definiert wurde. Beispiel: `Public Class Person` Die Verwendung von `Public` innerhalb von Prozeduren ist nicht möglich.

Tabelle 5.1 (Forts.) Übersicht der Zugriffsmodifier

AccessModifier	Bedeutung
`Protected`	`Protected` bedeutet, dass das so definierte Element innerhalb der Klasse, in der es definiert wurde, zugreifbar ist und in jeder von dieser Klasse abgeleiteten Klasse. Beispiel: `Protected mPersonalNummer As String` Die Verwendung von `Protected` ist nur innerhalb einer Klassendefinition, mit Ausnahme der Prozedurebene, möglich.
`Friend`	`Friend` bedeutet, dass das so definierte Element innerhalb der Assembly, in der es definiert wurde, zugreifbar ist. Ein Zugriff außerhalb dieser Assembly ist nicht möglich. Beispiel: `Friend mPersonalNummer As String` Die Verwendung von `Friend` auf Prozedurebene ist nicht möglich.
`Protected Friend`	`Protected Friend` ist eine Mischung aus `Protected` sowie `Friend` und bedeutet, dass das so definierte Element innerhalb der Assembly, in der es definiert wurde, zugreifbar ist und in jeder von der Klasse, in der es definiert wurde (auch außerhalb der Assembly), abgeleiteten Klasse. Beispiel: `Protected Friend mPersonalNummer As String` Die Verwendung von `Protected Friend` ist nur innerhalb einer Klassendefinition, mit Ausnahme der Prozedurebene, möglich.
`Private`	`Private` bedeutet, dass das so definierte Element nur innerhalb des Moduls, in dem es definiert wurde, zugreifbar ist. Ein Zugriff außerhalb dieses Moduls ist nicht möglich. Beispiel: `Private mPersonalNummer As String` Die Verwendung von `Private` auf Prozedurebene ist nicht möglich.
`Dim`	`Dim` bedeutet auf Modulebene dasselbe wie `Private`, Sie sollten aber die Definition mit `Private` vorziehen. Mit `Dim` sollten lokale Variablen auf Prozedurebene definiert werden. Beispiel: `Dim lokaleVariable As String` Eine Variable, die mit `Dim` innerhalb einer Prozedur in einem inneren Block (`If … End If`) definiert wurde, ist auch nur innerhalb dieses Blocks gültig.

Sämtliche Variablen sind so lange gültig, wie die Lebenszeit des übergeordneten Containers dauert. Das bedeutet, dass jede Variable, die auf Klassenebene definiert wurde, so lange gültig ist wie das Objekt, in dem es definiert wurde, Speicherplatz beansprucht. Lokale oder blockweite Variablen innerhalb von Prozeduren verlieren ihre Gültigkeit nach Abarbeitung der Prozedur oder des Blocks.

Achtung

Sämtliche auf Klassenebene definierten Variablen nennt man auch Instanzvariablen, da sie eine Gültigkeit in Abhängigkeit der ihr zugehörigen Instanz besitzen.

Shared Members

Es gibt auch die Möglichkeit, Variablen unabhängig von einer Objektinstanz zu definieren. Man spricht dabei von Klassenvariablen, denn der Wert dieser Variablen ist für alle Instanzen desselben Typs identisch. Solche Variablen werden mit dem zusätzlichen Modifier `Shared` definiert. Der Wert dieser Variablen kann direkt mit dem Klassennamen gesetzt oder ausgelesen werden. Shared Members werden in einem speziellen Speicherbereich unabhängig von den Klasseninstanzen gehalten. Ihre Gültigkeit erstreckt sich unabhängig von deren Existenz immer auf das ganze Programm.

Info

Es ist aus abwärtskompatiblen Gründen möglich, ein Shared Member über eine Objektvariable des entsprechenden Typs aufzurufen, jedoch sollten Sie dies grundsätzlich direkt über den Klassennamen machen. In VB.NET 1.1 war das ohne Probleme möglich, während es mittlerweile mit einer Compiler-Warnung quittiert wird. Ein Compiler-Fehler an dieser Stelle hätte die Abwärtskompatibilität gebrochen.

Mit dem Schlüsselwort `Shared` können auch Methoden von Klassen definiert werden, die somit aufgerufen werden können, ohne eine explizite Instanz eines Objekts anzulegen (oder haben Sie schon einmal ein Konsolenobjekt instanziiert, um `Console.WriteLine` aufzurufen?).

Beispiel für die Definition einer Shared-Variablen:

```
Public Class SharedBeispiel
    Public Shared Beruf As String
End Class
```

Auf diese Shared-Variable kann nun so zugegriffen werden:

```
Public Class UseOfShared
    Public Sub SetzeBeruf()
        SharedBeispiel.Beruf = "Taxifahrer"
    End Sub
End Class
```

Static-Variablen

Um die Lebenszeit von lokalen Variablen zu verlängern, gibt es die Möglichkeit, diese mit `Static` anstatt mit `Dim` zu definieren (obwohl man auch `Static Dim ...` schreiben könnte). Dies bedeutet, dass der reservierte Speicher

für die Variable nach Prozedurende nicht freigegeben wird. Das bedeutet auch, dass die Variable ihren Wert beim Beenden der Prozedur nicht verliert, sondern ihn über verschiedene Prozeduraufrufe hinweg beibehält.

Wenn eine `Static`-Variable innerhalb eines Moduls aufgerufen wird, besitzt die Variable einen Wert vom ersten Prozeduraufruf, mit dem sie definiert wurde, bis zum Programmende. Wird sie innerhalb einer Klasse in einer `Shared`-Methode definiert, erstreckt sich ihre Lebensdauer ebenfalls vom ersten Aufruf dieser Routine bis zum Ende des Programms. Wenn eine Variable innerhalb einer Klasse in einer normalen Methode (nicht Shared-Methode) deklariert wird, ist sie existent vom ersten Aufruf dieser Methode innerhalb der Instanz bis zur Speicherfreigabe dieser Instanz durch den Garbage Collector.

Friend Assembly

Friend Assembly ist eine Erweiterung der Sichtbarkeit von `Friend`-Klassen. Eine mit `Friend` definierte Klasse ist nur für Klassen innerhalb derselben Assembly sichtbar. Wird die Assembly in einer anderen Assembly referenziert, dann ist die `Friend`-Klasse für alle Klassen der anderen Assembly nicht sichtbar.

Sie können aber befreundete Assemblies definieren, die die Sichtbarkeit von `Friend`-Klassen eben genau auf diese befreundeten Assemblies ausweiten.

Eine Friend Assembly muss zwingend einen starken Namen, also eine digitale Signatur, besitzen. Nicht signierte Assemblies können nicht als befreundete Assemblies definiert werden.

Sie definieren eine Friend Assembly als Attribut, das für die gesamte Assembly gültig ist, also am besten innerhalb der *AssemblyInfo.vb*-Datei.

```
<Assembly: InternalsVisibleTo("OtherAssembly, _
           PublicKey=12345abcde012345")>
```

Eine Angabe der Versionsnummer oder Kulturinformation ist an dieser Stelle nicht zulässig.

5.1.6 Lebensdauer von Objekten

Die Lebensdauer von Objekten beginnt mit dem Zeitpunkt, zu dem das Objekt Speicher zugewiesen bekommt, und endet zu dem Zeitpunkt, an dem der Garbage Collector den Speicher für dieses Objekt wieder freigibt.

Dabei wird der Vorgang der Speicherzuweisung **Instanziierung** genannt, während die Speicherfreigabe auch als **Terminierung** bekannt ist.

Durch den `New`-Operator wird die Instanziierung durchgeführt, während der Garbage Collector ein zyklischer Prozess der Common Language Runtime ist, der nach nicht mehr referenzierten Objekten im Speicher sucht und diese dann zu einem nicht vorhersehbaren Zeitpunkt (nicht deterministisches Verhalten) freigibt.

Info

Anders als bei der Vorgängertechnologie COM wird bei .NET-Objekten kein Referenzzähler mitgeführt, der mit der Anweisung `Set object = Nothing` manipuliert werden konnte. War der Referenzzähler wieder bei 0 angekommen, sorgte das COM-Objekt aufgrund der COM-Standardschnittstellen für seine eigene Terminierung (zu einem vorhersehbaren Zeitpunkt).

Wird der `New`-Operator bereits bei der Definition eines Objekts mit angegeben, so wird das Objekt tatsächlich erst beim ersten Aufruf auch instanziiert. Ein Vorinstanziieren, und somit ein womöglich unnötiger Speicherverbrauch, findet dabei nicht statt.

Bei der Instanziierung wird innerhalb der Klasse der **Konstruktor** aufgerufen, während bei der Terminierung der **Destruktor** aufgerufen wird. Was diese beiden Spezialroutinen bewirken, erfahren Sie im folgenden Abschnitt.

5.1.7 Konstruktoren und Destruktoren

Konstruktoren und Destruktoren beinhalten die ersten und die letzten Codeanweisungen, die in einer Klasseninstanz ausgeführt werden. Wird eine Klasse mit `New` instanziiert, wird automatisch der Konstruktor der Klasse aufgerufen. Wenn zu einem späteren Zeitpunkt der Garbage Collector den Speicher für ein Objekt wieder freigibt, wird automatisch der Destruktor aufgerufen.

Konstruktor

Ein Konstruktor einer Klasse ist eine Sub und eignet sich hervorragend, um bestimmte Variablen einer Klasse vorzubelegen. Ein Konstruktor wird mit dem Schlüsselwort `New` definiert.

Wird innerhalb der Klasse kein Konstruktor angelegt, so legt der Compiler automatisch einen leeren sogenannten Default-Konstruktor an.

Im folgenden Konstruktor wird die Member-Variable der schreibgeschützten Property mit dem Wert 1000 vorbelegt.

```
Sub New()
    mGehalt = 1000
End Sub
```

Info

Der Konstruktor ist in etwa vergleichbar mit dem `Class_Initialize`-Ereignis aus VB 6. Jedoch konnten an dieses VB 6-Event keine Parameter übergeben werden.

Ein großer Vorteil von Konstruktoren besteht in der Tatsache, dass man in einer Klasse mehrere mit unterschiedlichen Signaturen (Anzahl und Datentypen der Parameter) implementieren kann. Diese Parameter sind die **Initialisierer** des Konstruktors. Das Überladen eines Konstruktors unterscheidet sich vom Overloading normaler Methoden dadurch, dass dabei das Schlüsselwort `Overloads` **nicht** verwendet werden darf.

Sehr oft implementiert man in Propertys Plausibilitäten, um zu vermeiden, dass Leerstrings oder Ähnliches übergeben werden. Das Problem dabei ist, dass der Getter nicht aufgerufen wird, solange der Eigenschaftswert nicht gesetzt wird. Folglich kann es trotzdem vorkommen, dass die Eigenschaft ein leerer String ist.

Um dem vorzubeugen, kann man einen Konstruktor definieren, an den die passenden Werte übergeben werden müssen.

Listing 5.10 Überladener Konstruktor

```
Sub New(ByVal personalNummer As String)
    Me.New()
    Me.PersonalNummer = personalNummer
End Sub
```

In Listing 5.10 wird zwingend die Übergabe einer Personalnummer verlangt. Um nicht redundanten Code schreiben zu müssen, kann man über die Instanzvariable `Me` den Standardkonstruktor, der bereits Logik enthält, aufrufen.

Um nun zu vermeiden, dass der Default-Konstruktor aufgerufen und somit wieder die Eingabe von Pflichtfeldern umgangen wird, kann man diesen Konstruktor auch mit `Private` oder `Protected` definieren und somit von außerhalb der Klasse unzugänglich machen. Es muss somit der Konstruktor mit dem Initialisierer aufgerufen werden.

Wird bei einem Konstruktor kein Modifier angegeben, so ist er standardmäßig `Public`.

Destruktor

Das Gegenteil zum Konstruktor stellt der Destruktor dar. Der Destruktor ist eine Routine, die automatisch von der Common Language Runtime ausgeführt wird, wenn der **Garbage Collector** den Speicherbereich für ein bestimmtes Objekt freigibt.

Innerhalb eines Destruktors sollten Aufräumarbeiten wie das Freigeben von benutzten Ressourcen und ähnliche Sachen durchgeführt werden.

Dadurch, dass der Destruktor automatisch von der CLR aufgerufen wird und von keinen anderen Klassen aufgerufen werden kann, muss er als `Protected` deklariert werden. Die `Sub Finalize` ist bereits in der Klasse `Object`, von der alle Klassen abgeleitet sind, implementiert, so dass diese Methode, wenn sie in einer Klasse implementiert wird, mit dem Schlüsselwort `Overrides` gekennzeichnet werden muss.

Es gibt außerdem nur einen einzigen Destruktor, nämlich einen parameterlosen Standarddestruktor. Es ist nicht zulässig, einen Destruktor mit Parametern zu erstellen.

Eine Implementierung eines Destruktors innerhalb einer Klasse würde folgendermaßen aussehen:

```
Protected Overrides Sub Finalize()
    'sauber machen
End Sub
```

Innerhalb eines Destruktors sollten Sie auf jeden Fall alle **unmanaged** Objekte (Objekte, die nicht von der Common Language Runtime verwaltet werden) freigeben, da die Speicherfreigabe dieser Objekte nicht vom Garbage Collector durchgeführt werden kann.

Achtung

Wundern Sie sich nicht, dass Ihnen `Finalize()` von IntelliSense nicht vorgeschlagen wird. Innerhalb des .NET Framework wird das `Dispose`-Muster bevorzugt, das wir uns gleich betrachten.

Dispose

Das Problem mit dem Destruktor unter .NET im Allgemeinen ist, dass er zu einem nicht vorhersehbaren Zeitpunkt aufgerufen wird. Das nennt man auch ein nicht deterministisches Verhalten. Das Objekt wird unter Umständen schon lange nicht mehr gebraucht, die `Finalize()`-Methode wurde jedoch noch nicht aufgerufen, weil der Garbage Collector das Objekt noch nicht freigegeben hat.

Will man bestimmte Ressourcen unabhängig vom Garbage Collector bereits zu einem früheren Zeitpunkt wieder freigeben, so hat man die Möglichkeit, das Interface `IDisposable` (mehr zu Interfaces im Abschnitt 5.3) zu implementieren.

Innerhalb der `Dispose()`-Methode dieses Interface kann man belegte Ressourcen freigeben, ohne dass man dazu auf den Garbage Collector warten muss. Der große Unterschied zur `Finalize()`-Methode besteht darin, dass die `Dispose()`-Methode nicht automatisch aufgerufen wird. Es ist nicht mehr als eine .NET-Konvention, die `Dispose()`-Methode aufzurufen, wenn das entsprechende Objekt nicht mehr benötigt wird. Ob das allerdings geschieht, hängt ganz im Gegensatz zur `Finalize()`-Methode vom Entwickler des Clients ab.

Im folgenden Code-Ausschnitt sehen Sie die Implementierung des Interface und der einzigen sich darin befindlichen Methode `Dispose()`.

```
Implements IDisposable
Public Sub Dispose() Implements IDisposable.Dispose
    'sauber machen
End Sub
```

Nun könnte man natürlich beide Möglichkeiten geschickt miteinander kombinieren. Wenn die `Dispose()`-Methode eines Objekts bereits aufgerufen wurde, was bedeutet, dass es sich um einen sauberen Cliententwickler handelt, dann will man natürlich nicht, dass der Finalizer noch einmal die Sachen freigibt. Deswegen kann man dem Garbage Collector über eine statische Methode `SuppressFinalize()` mitteilen, dass die Common Language Runtime nicht mehr den Destruktor aufruft. Wurde `Dispose()` nicht aufgerufen, so werden die Aufräumarbeiten durch die `Finalize()`-Methode angestoßen.

Der folgende Code-Ausschnitt illustriert diese Möglichkeit:

Listing 5.11
Zusammenspiel Dispose() und Finalize()

```
Implements IDisposable

Public Sub Dispose() Implements IDisposable.Dispose
    Dispose(True)
    GC.SuppressFinalize(Me)
End Sub
```

Listing 5.11 (Forts.) Zusammenspiel Dispose() und Finalize()

```
Protected Overridable Sub Dispose (ByVal disposing As Boolean)
    If disposing Then
        'gib managed Objekte frei
    End If
    ' gib unmanaged Objekte frei
End Sub

Protected Overrides Sub Finalize()
    Dispose(False)
End Sub
```

Hier wird in der `Dispose()`-Methode eine überladene `Dispose()`-Methode mit dem Parameter `True` aufgerufen und mit der statischen Methode `Supress-Finalize()` der Klasse `GC` die eigene Objektinstanz übergeben, um der Common Language Runtime mitzuteilen, dass die Aufräumarbeiten für dieses Objekt abgeschlossen sind und der Destruktor nicht mehr aufgerufen werden darf.

Im Finalizer wird auch die überladene Methode `Dispose()`, in der tatsächlich die Aufräumarbeiten stattfinden, aufgerufen. Ein Freigeben der verwalteten (managed) Objekte wäre an dieser Stelle doppelte Liebesmühe, denn die würde der Garbage Collector sowieso freigeben.

5.1.8 Vereinfachte Objektinitialisierung

Eine weitere Sprachneuerung, die vor allem die Produktivität erhöhen soll, ist die langersehnte Einführung der vereinfachten Objektinitialisierung.

Bei dieser Art der Objektinitialisierung können Sie mit dem Schlüsselwort `With` in geschweiften Klammern Initialwerte für die Eigenschaften einer Klasse setzen, ohne dafür explizit Konstruktoren zu definieren.

Listing 5.12 Vereinfachte Objektinitialisierung

```
Dim p As New Person() With {.Name = "Kotz",
  .Vorname = "Jürgen", .Geburtsdatum = New Date(1967, 12, 24)}
```

Im Beispiel in Listing 5.12 sehen Sie, wie eine Klasse vom Typ `Person`, welche aus den drei Eigenschaften `Name`, `Vorname` und `Geburtsdatum` besteht, durch die vereinfachte Objektinitialisierung ohne einen Konstruktor initialisiert wird.

Innerhalb von geschweiften Klammern, die dem Schlüsselwort `With` folgen, können Sie jeder Property mit einem zugänglichen `Set`-Zweig einen Wert zuweisen. Der Property muss syntaktisch ein Punkt voranstehen, wie Sie das vielleicht von dem `With`-Konstrukt bereits kennen.

Dabei kann jede mögliche Kombination von Zuweisungen erfolgen. Sie können, wie im Beispiel gezeigt, alle Eigenschaften vorinitialisieren, in jeglicher gewünschter Reihenfolge, Sie können nur einzelne Eigenschaften initialisieren oder jegliche zu wünschende Kombination auswählen.

In unserer kleinen Klasse mit nur drei Eigenschaften hätte man ansonsten bereits acht überladene Konstruktoren definieren müssen, um dies zu bewerkstelligen.

Sie dürfen jedoch diese Art der Objektinitialisierung keinesfalls als einen Ersatz für einen Konstruktor sehen, denn Konstruktoren sind natürlich auch weiterhin in dieser Konstellation möglich. Sie können auch weiter einen oder mehrere überladene Konstruktoren, in denen Sie natürlich auch weiterhin private Felder initialisieren können, definieren und mit der vereinfachten Objektinitialisierung kombinieren. Es wird aber definitiv erst der Konstruktor abgearbeitet und erst danach werden die entsprechenden `Set`-Zweige der zu initialisierenden Properties ausgeführt.

Auch an dieser Stelle werden Sie wieder von IntelliSense in Visual Studio unterstützt, wie Sie in Abbildung 5.2 sehen.

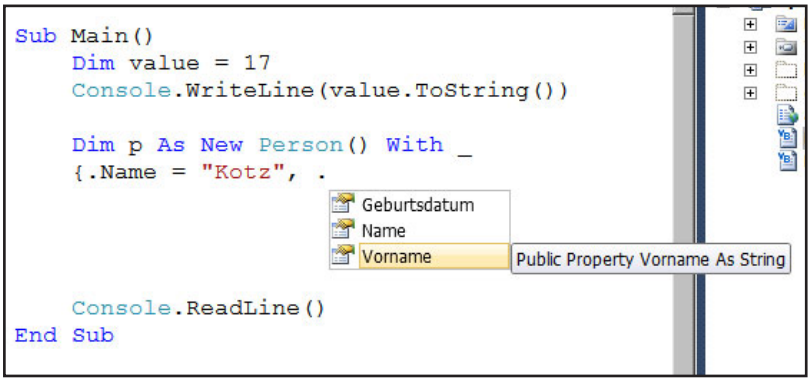


Abbildung 5.2
IntelliSense bei vereinfachter Objektinitialisierung

5.1.9 Partielle Klassen

Partielle Klassen, oder auch **Partial Classes**, sind eine Erweiterung, die mit dem .NET Framework 2.0 eingeführt wurde. Mittels dieser partiellen Klassen kann man eine Klassendefinition in zwei voneinander getrennten physikalischen Dateien speichern. Somit kann zum Beispiel automatisch generierter Designercode von eigenem geschriebenen Code getrennt werden, was Visual Studio 2010 auch macht.

Es können dabei beliebig viele partielle Klassen, und somit auch physikalische Dateien, definiert werden, die Klassenteile müssen lediglich alle innerhalb derselben **Assembly** und innerhalb des gleichen **Namespace** stehen.

Alle partiellen Klassen müssen außerdem denselben Gültigkeitsbereich, wie zum Beispiel `Public` oder `Protected`, besitzen. Das Schlüsselwort `partial` muss dabei mindestens bei einer Definition angegeben werden.

Die ursprüngliche Idee von partiellen Klassen war eine konsequente Trennung von Designercode und eigenem Entwicklercode. Deshalb sollte man selbst partielle Klassen nur für diese Trennung benutzen und nicht einfach, weil es ein besonderes Feature ist.

Eine partielle Klasse wird wie folgt definiert:

```
Partial Public Class Classx
```

Das Schlüsselwort `partial` kann nur für Klassen oder die in Abschnitt 4 beschriebenen **Strukturen** verwendet werden. Es ist für andere Konstrukte (z.B. Interfaces) nicht zulässig.

5.1.10 Partielle Methoden

Partielle Klassen wurden in der Version 2.0 des .NET Framework eingeführt und sollen eine saubere Trennung von Designercode und eigenem Code gewährleisten. In der Nachfolgeversion .NET Framework 3.5 und somit auch in der Sprache Visual Basic 9 folgte die Einführung von partiellen Methoden.

Der Sinn von partiellen Methoden ist derselbe wie der von partiellen Klassen, nämlich eine saubere Trennung von Designercode und selbst geschriebenem Code. Jedoch ist die Implementierung von partiellen Methoden komplett anders.

Voraussetzung für die Implementierung von partiellen Methoden sind partielle Klassen. Das bedeutet: Mittels partieller Methoden kann eine Methodendefinition in zwei unterschiedlichen physikalischen Dateien stehen. Eine Aufteilung auf mehrere physikalische Dateien ist jedoch nicht möglich und auch nicht sinnvoll.

An einer Stelle, in der Regel innerhalb des Designercodes, steht lediglich die Definition der Methode.

```
Partial Private Sub OnNameChanged()
End Sub
```

In diesem kleinen Codebeispiel wurde der Modifier `Private` nicht zufällig gewählt. Partielle Methoden können ausschließlich als `Private` definiert werden.

Im zweiten Teil der partiellen Klasse steht dann dieselbe Methode mit der kompletten Implementierung. An dieser Stelle darf das Schlüsselwort `Partial` jedoch nicht mehr verwendet werden.

```
Private Sub OnNameChanged()
   Console.WriteLine("Der Name hat sich verändert")
End Sub
```

So weit, so gut, die Syntax ist wohl ziemlich klar. Aber wo liegt der Sinn dieser Sache?

Um nicht zu viel vorzugreifen: Es gibt in LINQ auch Designer, die Klassendefinitionen aus Datenbankinhalten erstellen. Innerhalb dieser Designerklassen werden Sie in den Properties Aufrufe von partiellen Methoden im `Set`-Zweig finden. Es wird also eine Methode aufgerufen, wenn sich der Wert einer Property ändert. Was jedoch genau und ob überhaupt etwas passieren soll, das kann der Designer natürlich nicht entscheiden. Deswegen fügt der Designer lediglich einen mit dem Schlüsselwort `Partial` gekennzeichneten Methodenrumpf in die Klasse ein und ruft diese Methode in der entsprechenden Property auf.

Der Entwickler kann jetzt in einer anderen physikalischen Datei die Logik für die partielle Methode implementieren. Das bringt den Vorteil, dass er keine Angst haben muss, dass bei einem Neugenerieren der Klasse durch den Designer seine Logik überschrieben wird. Implementiert der Entwickler die partielle Methode nicht in seinem eigenen Code, so erkennt das der Compiler und ignoriert den Aufruf der partiellen Methode sowie den Methodenrumpf.

Das folgende Beispiel soll das noch einmal illustrieren, wobei Listing 5.13 den Inhalt der Designerdatei darstellen soll und Listing 5.14 den selbst geschriebenen Programmcode.

Listing 5.13
Partielle Methoden in einer Designerdatei

```
Partial Public Class Person

    Partial Private Sub OnNameChanged()
    End Sub

    Partial Private Sub OnVornameChanged()
    End Sub

    Partial Private Sub OnGeburtsdatumChanged()
    End Sub

    Private mname As String
    Public Property Name() As String
        Get
            Return mname
        End Get
        Set(ByVal value As String)
            mname = value
            OnNameChanged()
        End Set
    End Property

    Private mvorname As String
    Public Property Vorname() As String
        Get
            Return mvorname
        End Get
        Set(ByVal value As String)
            mvorname = value
            OnVornameChanged()
        End Set
    End Property

    Private mgeburtsdatum As DateTime
    Public Property Geburtsdatum() As DateTime
        Get
            Return mgeburtsdatum
        End Get
        Set(ByVal value As DateTime)
            mgeburtsdatum = value
            OnGeburtsdatumChanged()
        End Set
    End Property

End Class
```

Listing 5.14 Partielle Methoden in der eigenen Codedatei

```
Partial Public Class Person

    Private Sub OnNameChanged()
        Console.WriteLine("Der Name hat sich verändert")
    End Sub

    Private Sub OnVornameChanged()
        Console.WriteLine("Der Vorname hat sich verändert")
    End Sub

    Private Sub OnGeburtsdatumChanged()
       Console.WriteLine("Das Geburtsdatum hat sich verändert")
    End Sub

End Class
```

5.1.11 Anonyme Typen

Wie schon im Abschnitt zu impliziter Typisierung angekündigt, gibt es mit LINQ Funktionen, deren Rückgabewerte bislang nicht definierte Typen darstellen. Um dies zu realisieren, wurden anonyme Typen eingeführt.

Anonyme Typen bieten somit die Möglichkeit, komplexe Typen zu verwenden, ohne dafür eine Klasse zu definieren.

Bei der Definition einer Variablen für einen anonymen Typ ist wiederum das `With`-Konstrukt von Bedeutung.

Listing 5.15 Definition eines anonymen Typs

```
Dim veranstaltung = New With {.Name = "CeBit",
        .Datum = New DateTime(2010, 3, 4)}
```

In Listing 5.15 sehen Sie, wie ein anonymer Typ mit zwei Eigenschaften `Name` und `Datum` definiert wird. Aufbauend auf impliziter Typisierung wird auf die Angabe des Datentyps, der ja nicht bekannt ist, verzichtet. Dem `New`-Operator folgt dabei das Schlüsselwort `With` und genauso wie bei der vereinfachten Objektinitialisierung werden dann die Properties mit Werten versorgt. Auch hier erfolgt wiederum implizite Typisierung durch die Wertzuweisung, da ansonsten der Datentyp für die Properties `Name` und `Datum` nicht zu ermitteln wäre.

Das bedeutet bei der Definition eines anonymen Typs, dass keine weiteren Klassenmember, bis auf die von `Object` geerbten, möglich sind. Es sind auch keine Methodendefinitionen bei anonymen Typen möglich, sondern lediglich die Angabe von Properties. Die angegebenen Properties müssen auch sofort initialisiert werden und sie unterliegen auch sonst den gleichen Regeln wie bei der impliziten Typisierung.

Dadurch, dass anonyme Typen auf implizite Typisierung aufbauen, können diese auch nur lokal verwendet werden.

Wenn Sie wiederum einen Blick auf den IL-Code werfen, werden Sie sehen, dass der Compiler einen Klassennamen generiert. Dieser kann jedoch direkt im Code nicht verwendet werden.

Und auch an dieser Stelle gibt es in Visual Studio wiederum volle IntelliSense-Unterstützung, wie Abbildung 5.3 zeigt.

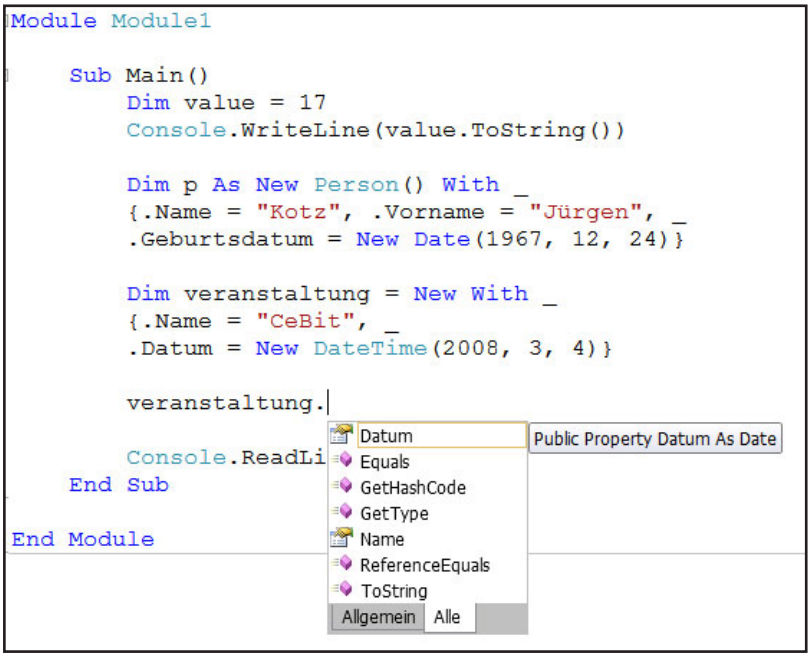


Abbildung 5.3
IntelliSense bei anonymen Typen

Genauso wie bei der impliziten Typisierung ist auch an dieser Stelle wieder darauf hinzuweisen, dass anonyme Typen vor allem für das Zusammenspiel mit LINQ implementiert wurden und nicht an allen beliebigen Stellen verwendet werden sollten.

5.2 Vererbung

Mittels **Vererbung**, die in der ersten Version von VB.NET eingeführt wurde, stieg Visual Basic zu einer wirklichen objektorientierten Sprache auf.

Vererbung bedeutet, dass eine neue Klasse alle Member einer bereits bestehenden Klasse (Basisklasse) automatisch enthält und zu dieser Klasse typenkompatibel ist. Eine erneute Implementierung der in der Basisklasse bereits definierten Klassen-Member ist nicht mehr nötig. Anschließend können zu dieser neuen Klasse neue Eigenschaften und Methoden hinzugefügt oder auch bestehende Member überschrieben werden. Die Basisklasse kann somit in der neuen, erbenden Klasse spezialisiert werden.

Info

In anderen Programmiersprachen spricht man dabei nicht von einer Basisklasse, sondern von einer Superklasse.

In welcher .NET-Sprache dabei die Basisklasse geschrieben wurde, ist vollkommen egal, denn vererbt wird, wenn die Klasse in einem anderen Projekt liegt, von der bereits kompilierten Assembly, die im Format der unabhängigen **Intermediate Language** (IL) vorliegt.

Somit bietet sich auch die Möglichkeit, von den .NET-Basisklassenbibliotheken zu erben, was viele Aufgaben vereinfacht. Auf diese Weise kann man zum Beispiel bequem eigene Exception-Klassen für eine spezifische Fehlerbehandlung definieren oder auch bestehende Steuerelemente mit zusätzlicher Funktionalität erweitern.

Vererbung (**Inheritance**) wird in Visual Basic 10 mit dem Schlüsselwort `Inherits` implementiert. Wollen Sie von einer Klasse erben, so muss sofort nach dem Beginn der Klassendefinition, als erste Anweisung, die `Inherits`-Anweisung (mit einem Verweis auf die Basisklasse) stehen.

```
Public Class Mitarbeiter
    Inherits Person
End Class
```

Obwohl diese Anweisung sehr einfach ist, erweist sich Vererbung als ein sehr mächtiges Konzept, was zu einem viel wartbareren, strukturierten und übersichtlichen Programmcode führt.

Wenn Sie nicht wollen, dass von einer Klasse abgeleitet wird, kann man die Vererbung für diese Klasse mit dem Schlüsselwort `NotInheritable` verbieten.

```
Public NotInheritable Class Something
```

In Kapitel 4.1.2 »Werfen einer Ausnahme« wurde darauf hingewiesen, wie einfach es mittels Vererbung ist, eigene `Exception`-Typen zu definieren. Das folgende Beispiel zeigt die Definition einer eigenen Personen-Exception, die natürlich von der Basisklasse `Exception` abgeleitet wird. Für diese neue Klasse kann man einen eigenen `Catch`-Zweig für eigene ausgelöste Fehler in der Klasse implementieren.

Listing 5.16
Beispiel für eine eigene Exception-Klasse

```
Public Class PersonException
    Inherits Exception
    Sub New()
        MyBase.New ("Fehler in der Personenklasse")
    End Sub
    Sub New(ByVal Message As String)
        MyBase.New(Message)
    End Sub
End Class
```

In diesem Beispiel wird eine neue Klasse `PersonException` definiert, die von der Basisklasse `Exception` abgeleitet wurde. Diese Klasse wurde mit zwei überladenen Konstruktoren implementiert. Wird der parameterlose Konstruktor aufgerufen, dann wird eine Standardmeldung an den Konstruktor der Basisklasse übergeben, um die Eigenschaft `Message` der Exception-Klasse zu setzen. Im zweiten Konstruktor kann eine frei wählbare Exception-Meldung übergeben werden, die wiederum an den Basisklassenkonstruktor übergeben wird. Schon ist unsere neue Exception-Klasse fertig.

5.2.1 Einfachvererbung

Innerhalb des .NET Framework ist nur Einfachvererbung möglich. Das bedeutet, dass nur von einer einzigen Basisklasse abgeleitet werden kann. In anderen Programmiersprachen, wie zum Beispiel C++, ist auch Mehrfachvererbung möglich, das Erben aus verschiedenen Basisklassen. (Zum Beispiel könnte ein Amphibienfahrzeug von Boot und Automobil erben.)

Einfachvererbung bedeutet somit, dass die Basisklasse eindeutig ist. Diese kann natürlich über mehrere Stufen hinweg weitervererbt werden.

Mehrfachvererbung birgt die Gefahr unübersichtlichen Codes in sich, weshalb sie viele Entwickler auch in C++ gar nicht benutzen. Dies ist wohl auch der Grund, warum es innerhalb des .NET Framework nur Einfachvererbung gibt.

5.2.2 Besonderheit des Konstruktors bei der Vererbung

Das einzige Element, das von der Basisklasse nicht vererbt wird, ist der Konstruktor. Jedoch wird jeder Konstruktor der Basisklasse(n) aufgerufen, angefangen bei der Wurzelklasse.

Das Konsolenbeispielprogramm in Listing 5.17 soll diesen Mechanismus verdeutlichen:

Listing 5.17 Beispielprogramm Konstruktormechanismus bei vererbten Klassen

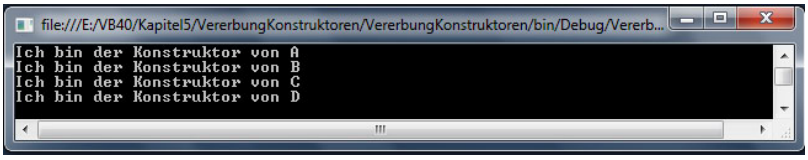
```
Module Module1
    Sub Main()
        Dim test As New D
        Console.ReadLine()
    End Sub
End Module

Public Class A
    Sub New()
        Console.WriteLine ("Ich bin der Konstruktor von A")
    End Sub
End Class

Public Class B
    Inherits A
    Sub New()
        Console.WriteLine ("Ich bin der Konstruktor von B")
    End Sub
End Class

Public Class C
    Inherits B
    Sub New()
        Console.WriteLine ("Ich bin der Konstruktor von C")
    End Sub
End Class
```

Listing 5.17 (Forts.) Beispielprogramm Konstruktormechanismus bei vererbten Klassen

```
Public Class D
    Inherits C
    Sub New()
        Console.WriteLine ("Ich bin der Konstruktor von D")
    End Sub
End Class
```

In diesem kleinen Beispielprogramm gibt es vier Klassen A, B, C und D, wobei B von A, C von B und D von C erbt. Somit erbt D tatsächlich von C, B und A. In den jeweiligen Konstruktoren wird eine Meldung in der Konsole ausgegeben, dadurch sieht man, in welcher Reihenfolge diese Konstruktoren aufgerufen werden. (Aber bitte kommen Sie in Ihren eigenen Klassen nicht auf die Idee, im Konstruktor ähnliche Ausgaben zu produzieren!!!) Wird nun D instanziiert, werden sämtliche Konstruktoren der vier Basisklassen A, B, C und D aufgerufen, was Abbildung 5.4 auch zeigt.

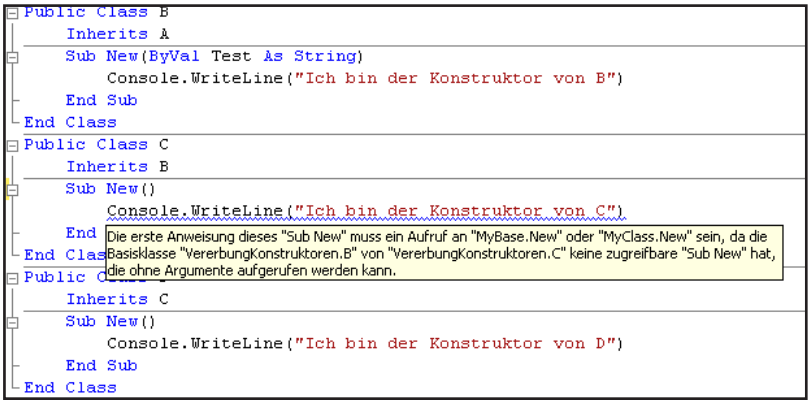
Abbildung 5.4 Konstruktorausgaben



So weit, so gut, aber was passiert, wenn jetzt in einer Klasse, zum Beispiel in der Klasse C, kein Konstruktor implementiert wurde? Dann erzeugt der Compiler automatisch einen Default-Konstruktor, die Aufrufe gehen wie gehabt die Vererbungshierarchie hoch. Der Compiler erzeugt jedoch nur einen Standardkonstruktor, wenn Sie selbst keinen Konstruktor angelegt haben.

Interessant wird es jetzt aber, wenn kein Default-Konstruktor in einer Klasse vorhanden ist, weil überladene Konstruktoren vorhanden sind. Woher soll der Compiler wissen, welcher Konstruktor beim Instanziieren eines vererbten Objekts aufgerufen werden soll? Solange ein parameterloser Konstruktor vorhanden ist, wird grundsätzlich dieser aufgerufen. Wurde ein Konstruktor mit Parametern, jedoch kein Standardkonstruktor definiert, dann würde der Compiler einen Fehler melden, denn er wüsste nicht, welche Werte er an den nicht parameterlosen Konstruktor übergeben soll.

Abbildung 5.5 Compiler-Fehler aufgrund eines fehlenden zugreifbaren Konstruktors



Um dieses Problem zu umgehen, kann man mit der Objektvariablen `MyBase` direkt einen spezifischen Konstruktor der Basisklasse aufrufen, so dass man die Wahl hat, welcher der überladenen Konstruktoren bei der Instanziierung aufgerufen wird, und diesem auch plausible Werte übergeben kann. `MyBase` steht dabei direkt für die Basisklasse, von der die aktuelle Klasse abgeleitet wurde.

Listing 5.18
Aufruf von MyBase.New()

```
Public Class C
    Inherits B
    Sub New()
        MyBase.New("Test")
        Console.WriteLine ("Ich bin der Konstruktor von C")
    End Sub
End Class
```

Dabei muss die Anweisung `MyBase.New()` die erste Anweisung innerhalb des Konstruktors sein.

Sie können auf diese Weise natürlich verschiedene Überladungen des Basisklassenkonstruktors aufrufen.

5.2.3 Besonderheiten bei der Vererbung von Events

Events werden zwar mitvererbt, jedoch gibt es keine Möglichkeit, aus der abgeleiteten Klasse direkt den Event auszulösen. Wird eine vererbte Methode über die abgeleitete Klasse aufgerufen, so wird der Event problemlos ausgelöst und im Client kann auch ohne Weiteres darauf reagiert werden.

Will man jedoch in einer zusätzlichen Methode oder Property den Event auslösen, dann meckert der Compiler, wie man in Abbildung 5.6 sieht, mit der Meldung, dass Basisklassen-Events nicht direkt aus vererbten Klassen getriggert werden können.

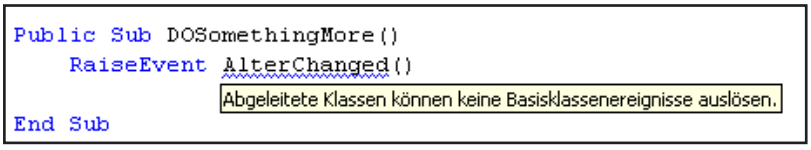


Abbildung 5.6
Compiler-Fehler beim direkten Aufruf eines Events

Um dieses Problem zu lösen, kann man in der Basisklasse eine interne (`Protected`-)Routine schreiben, die den entsprechenden Event auslöst.

```
Protected Sub CallEventAlterChanged
   (ByVal sender As Object, ByVal e As EventArgs)

    RaiseEvent AlterChanged(sender, e)
End Sub
```

Die entsprechende Prozedur kann dann aus der vererbten Klasse mittels der Objektvariablen `MyBase` aufgerufen werden.

```
MyBase.CallEventAlterChanged(Me, New EventArgs())
```

Ansonsten ist bei der Vererbung von Events nichts weiter zu beachten.

5.2.4 Typenkompatibilität

Jede abgeleitete Klasse ist zur Basisklasse typenkompatibel, aber nicht umgekehrt. Jede abgeleitete Klasse besitzt dieselben Schnittstellen wie die Basisklasse, da diese ja sämtlich vererbt werden. Jedoch fügt man in der abgeleiteten Klasse in der Regel zusätzliche Schnittstellen hinzu und deshalb kann die Basisklasse nicht mehr typenkompatibel zur abgeleiteten Klasse sein, da genau diese neuen Schnittstellen fehlen.

Durch die Typenkompatibilität können Sie eine Objektvariable vom Typ der Basisklasse definieren und zu einem späteren Zeitpunkt als Typ der abgeleiteten Klasse instanziieren.

```
Dim MA As Person
MA = New Mitarbeiter("2606-C80-162")
```

Wenn mehrere unterschiedliche abgeleitete Klassen vorhanden sind, deren Typ erst während der Laufzeit erkannt wird, gibt einem diese Typenkompatibilität sehr viele Möglichkeiten, seinen Code übersichtlich und kompakt zu halten, ohne für jeden unterschiedlichen Typ eine eigene Objektvariable vorzuhalten.

5.2.5 Sprachübergreifende Vererbung

Wie schon einleitend erwähnt, ist es im .NET Framework völlig egal, mit welcher .NET-Sprache die Basisklasse ursprünglich programmiert wurde. Da alle Sprachen zu einem einheitlichen Zwischencode, der Microsoft Intermediate Language, kompiliert werden, ist es zu diesem Zeitpunkt gar nicht ersichtlich, in welcher Sprache die Assembly entwickelt wurde. Wenn man sprachübergreifend vererben will, muss die Assembly, in der die Basisklasse enthalten ist, als Referenz in das Projekt eingebunden werden. Die in einer anderen Sprache programmierte Klasse kann nicht im selben Projekt liegen, da innerhalb eines Projekts nur eine Programmiersprache verwendet werden kann.

Nachdem ein Projektverweis auf die Assembly, in der die Klasse liegt, von der geerbt werden soll, gesetzt wurde, kann man von der gewünschten Basisklasse ableiten, als wäre sie im selben Projekt und in derselben Sprache geschrieben. Der einzige Unterschied liegt darin, dass der Code der Klasse nicht sichtbar ist. Um sich zumindest einen Überblick über sämtliche Schnittstellen der Klasse zu verschaffen, kann man sich diese im Objektkatalog anschauen.

Den Objektkatalog können Sie über das Menü Ansicht/Objektkatalog aufrufen.

Hier sehen Sie ein Beispiel für die Klasse `FileInfo` im Namespace `System.IO`. Im rechten Fenster des Objektkatalogs sehen Sie dabei alle Klassen-Member mit einer kurzen Beschreibung und der zugehörigen Parameterauflistung in einem darunterliegenden Fenster.

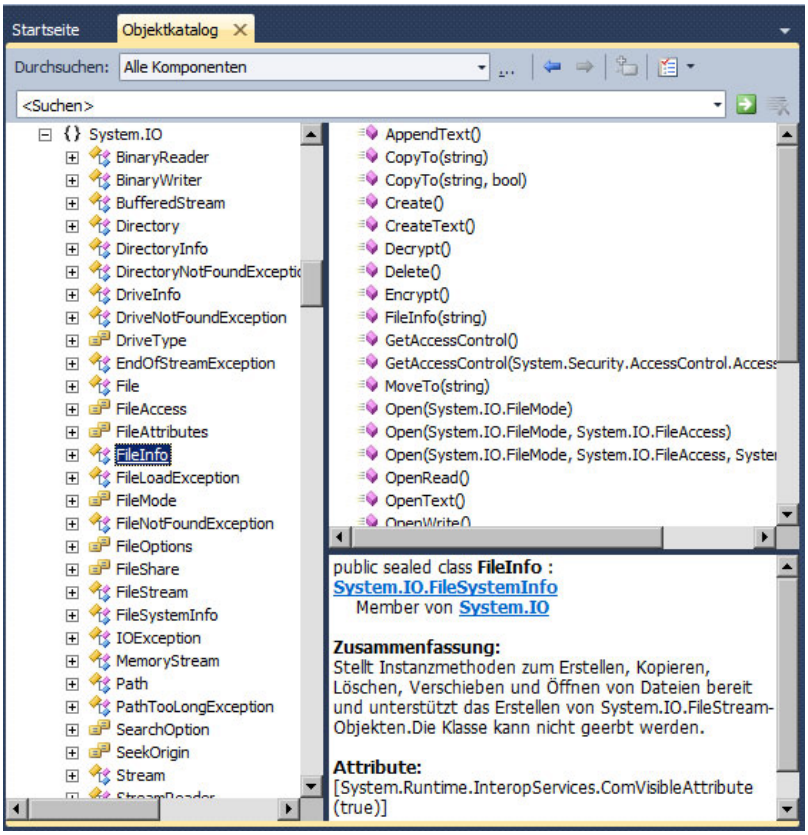


Abbildung 5.7
Objektkatalog

5.2.6 Überschreiben von Membern

Wenn eine Klasse durch Vererbung spezialisiert werden soll, muss auch die Möglichkeit bestehen, vorhandene Funktionalität in der neuen Klasse anzupassen. Dies wird durch das Überschreiben (*Overriding*) von Eigenschaften und Methoden gewährleistet. Um ein Member in einer abgeleiteten Klasse zu überschreiben, muss dieses Member in der Basisklasse auch als überschreibbar gekennzeichnet werden. Wenn das nicht der Fall ist, würde der Compiler mit einer Fehlermeldung reagieren, wenn man versucht, das Member zu überschreiben. Standardmäßig sind Member nicht überschreibbar.

Wird das Member nun überschrieben, dann besitzt die abgeleitete Klasse eine neue Funktionalität für die weiterhin kompatible Schnittstelle.

Um ein Member in einer Basisklasse als überschreibbar zu kennzeichnen, muss es mit dem Schlüsselwort `Overridable` versehen werden. Dieses Member kann danach in der abgeleiteten Klasse mit dem Schlüsselwort `Overrides` überschrieben werden.

Ist ein Member einmal mit `Overridable` gekennzeichnet, kann diese Methode in allen von dieser Klasse abgeleiteten Unterklassen, egal in welcher Hierarchietiefe, überschrieben werden. Mit dem Schlüsselwort `NotOverridable` kann man die Überschreibbarkeit wieder aufheben.

In Listing 5.19 (Code der Basisklasse) und Listing 5.20 (Code der abgeleiteten Klasse) wird gezeigt, wie *Overriding* implementiert werden kann.

Listing 5.19 Funktion als überschreibbar definieren

```
Public Class Person
  Public Overridable Function BerechneGehalt() As Double
        Return 2000
    End Function
…
End Class
```

Hier wird die Funktion `BerechneGehalt()` als überschreibbar gekennzeichnet.

In Listing 5.20 wird diese Funktionalität nun überschrieben.

Listing 5.20 Überschreiben der Basisklassenmethode

```
Public Class Angestellter
    Inherits Person

    Public NotOverridable Overrides Function
       BerechneGehalt() As Double
        Return MyBase.BerechneGehalt() + BerechneBonus()
    End Function

    Public Function BerechneBonus() As Double
        Dim Bonus As Double = 10
        'Algorithmus für Bonusberechnung
        Return Bonus
    End Function
…
End Class
```

In der Klasse `Angestellter` wird die Funktionalität `BerechneGehalt()` überschrieben. Dabei wird auf das Ergebnis, das die Basisklassenberechnung ergibt, noch ein Bonus aufaddiert, der in dieser Klasse `Angestellter` berechnet wird. Die ursprüngliche Berechnungsroutine der Basisklasse kann jederzeit über `MyBase` aufgerufen werden. Durch das Schlüsselwort `NotOverridable` wird es Klassen, die von `Angestellter` abgeleitet werden, verboten, diese Methode wiederum zu überschreiben.

In einer weiteren Klasse `Arbeiter` wird die Funktionalität anders überschrieben.

Listing 5.21 Weitere Überschreibung

```
Public Class Arbeiter
    Inherits Person
    Private mStunden As Single
    Public Property Stunden() As Single
        Get
             Return mStunden
        End Get
        Set(ByVal value As Single)
            mStunden = value
        End Set
    End Property
```

```
    Public Overrides Function BerechneGehalt() As Double
        Return Stunden * 17
    End Function
…
End Class
```

Listing 5.21 (Forts.)
Weitere Überschreibung

In dieser Klasse wird die Funktionalität der Basisklasse mit einer anderen Logik überschrieben. Hier wird auf die Basisklassenfunktionalität überhaupt nicht zugegriffen, sondern es wird eine komplett neue Logik aufgesetzt. Da das Schlüsselwort `NotOverridable` hier nicht gesetzt wurde, kann jede Klasse, die von Arbeiter erbt, diese Funktion auch wiederum überschreiben.

Wird nun eine Instanz vom Typ `Person` erzeugt, so wird die Funktionalität ausgeführt, wie sie in der Basisklasse implementiert wurde. Wird das Objekt vom Typ `Angestellter` oder `Arbeiter` instanziiert, so wird die jeweils in der entsprechenden Klasse implementierte Funktionalität aufgerufen.

So weit, so gut, was aber passiert jetzt, wie es in der Klasse `Angestellter` der Fall ist, wenn die überschriebene Methode mittels `MyBase` auch die Funktionalität der Basisklasse aufruft und hier wiederum eine Methode aufgerufen wird, die in der abgeleiteten Klasse ebenfalls überschrieben wurde?

Das folgende Beispiel soll dies illustrieren (Listing 5.22 und Listing 5.23).

Betrachten Sie zuerst den abgeänderten Code der Basisklasse in Listing 5.22:

```
Public Class Person

  Public Overridable Function BerechneGehalt() As Double
        Return 2000 – BerechneAbzüge()
  End Function

  Public Overridable Function BerechneAbzüge() As Double
        Return 500
    End Function
…
End Class
```

Listing 5.22
Code der Basisklasse

Die Methode `BerechneGehalt()` ruft jetzt intern eine ebenso überschreibbare Methode `BerechneAbzüge()` auf, um das Gehalt zu berechnen.

In der abgeleiteten Klasse `Angestellter` wird nun in Listing 5.23 genau diese Methode ebenso überschrieben.

```
Public Class Angestellter
    Inherits Person

    Public NotOverridable Overrides Function
      BerechneGehalt() As Double
        Return MyBase.BerechneGehalt() + _
               BerechneBonus()
    End Function

    Public Overrides Function BerechneAbzüge() As Double
        Return 500 + BerechneBonus() * 0.1
    End Function
```

Listing 5.23
Code der abgeleiteten Klasse

Listing 5.23 (Forts.) Code der abgeleiteten Klasse

```
    Public Function BerechneBonus() As Double
        Dim Bonus As Double = 10
        'Algorithmus für Bonusberechnung
        Return Bonus
    End Function
…
End Class
```

Da der Aufruf von `BerechneGehalt()` jetzt von der Basisklasse aus und in dieser Methode die Funktion `BerechneAbzüge()` aufgerufen wird, stellt sich die Frage, welche Funktionalität nun zum Zuge kommt.

Wird die Methode über eine Objektvariable aufgerufen, die vom Typ `Person` instanziiert wurde, ist es wohl völlig klar, dass die `BerechneAbzüge()`-Funktionalität der Basisklasse aufgerufen wird, was zu einem Ergebnis von 1500 führt.

Wenn nun die Methode über eine Objektvariable vom Typ `Angestellter` aufgerufen wird, dann wird aufgrund von **Polymorphie** (mehr dazu in Abschnitt 3) die Funktionalität von `BerechneAbzüge()` ausgeführt, so wie sie in der abgeleiteten Klasse `Angestellter` implementiert wurde, was zu einem Ergebnis von 1509 (2000 + 10 – 500 – 10 * 0.1) führt.

Listing 5.24 Testprogramm zur Ausgabe der unterschiedlichen Gehälter

```
Public Module Module1
    Sub Main()
        Dim x As New Person
        Dim y As New Angestellter
        Console.WriteLine _
           ("Gehalt einer Person: " &
             x.BerechneGehalt().ToString)
        Console.WriteLine("Gehalt eines Angestellten: " &
            y.BerechneGehalt().ToString)
        Console.ReadLine()
    End Sub
End Module
```

Diese Aussage wird belegt durch die Bildschirmausgabe in Abbildung 5.85.8.

Abbildung 5.8 Ausgabe des Testprogramms

Wenn man jedoch will, dass beim Aufruf an die Basisklasse alle weiteren Aufrufe auch innerhalb der Basisklasse bleiben, so kann man dies mittels des Schlüsselworts `MyClass` bewerkstelligen.

Wenn beim Aufruf von `BerechneAbzüge()` in der Basisklasse `MyClass` als Kontext vorangesetzt wird, so bleibt der Aufruf innerhalb der Basisklasse und geht nicht an die abgeleitete Klasse `Angestellter` zurück.

```
Public Overridable Function BerechneGehalt() As Double
    Return 2000 - MyClass.BerechneAbzüge()
End Function
```

Durch diese kleine Codeänderung ändert sich jetzt die Berechnung für den Angestellten auf 1510 (2000 + 10 – 500), was zur Bildschirmausgabe in Abbildung 5.9 führt.

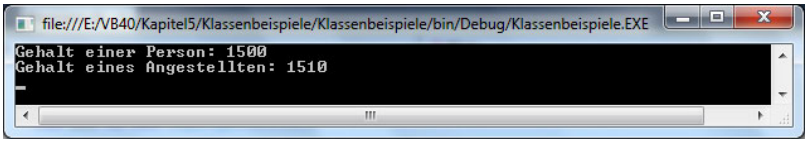


Abbildung 5.9
Ausgabe des Testprogramms mit MyClass

Das Überschreiben von Members ist ein wichtiges Merkmal zur Spezialisierung von Klassen. Auf den vorstehenden Seiten wurden die Möglichkeiten, die beim *Overriding* zur Verfügung stehen, aufgezeigt. In diesen Beispielen wurde auch zum ersten Mal der Begriff der Polymorphie erwähnt. Um Polymorphie abschließend zu erläutern, will ich im nächsten Kapitel noch zwei weitere Kerntechnologien der objektorientierten Programmierung und der Polymorphie, nämlich **Interfaces** und **abstrakte Basisklassen**, einführen.

5.3 Interfaces und abstrakte Basisklassen

Interfaces und abstrakte Basisklassen sind wesentliche Technologien der objektorientierten Programmierung. Beide setzen dabei auf Typenkompatibilität durch den Einsatz von fest definierten Schnittstellen, die mit unterschiedlicher Funktionalität hinterlegt sind.

Der Aufruf von identischen Schnittstellen aus unterschiedlichen Kontexten führt dabei zur Ausführung von unterschiedlicher Funktionalität. Dies wird als Polymorphie (Vielgestaltigkeit) bezeichnet, da gleiche Aufrufe unterschiedliche Ergebnisse liefern können.

Durch Interfaces wird ein Satz von Schnittstellen definiert, eine Implementierung von Logik ist nicht möglich. Abstrakte Basisklassen dagegen sind eine Mischung aus einer Definition von Schnittstellen und der Implementierung von realen Funktionalitäten. Eine abstrakte Basisklasse selbst kann jedoch nicht instanziiert, sondern lediglich vererbt werden. In der vererbten Klasse müssen dann die definierten Schnittstellen implementiert werden.

5.3.1 Interfaces

Interfaces, zu Deutsch Schnittstellen, haben unterschiedlichste Einsatzmöglichkeiten innerhalb ihrer Applikationen. Das wesentliche Ziel, das mit Interface-Definitionen erreicht werden soll, ist eine Kompatibilität mit einem bestimmten Satz von Schnittstellen, die in unterschiedlichsten Klassen implementiert werden können.

Somit können Typen, die eigentlich nichts miteinander zu tun haben (jedoch dasselbe Interface implementieren), innerhalb von Routinen identisch behandelt werden, da sie für diese Schnittstellen kompatible Aufrufe besitzen.

Das Interface ist dabei nur eine Vorschrift für die implementierende Klasse, alle Schnittstellen des Interface innerhalb dieser zu definieren. Dabei können unterschiedlichste Logiken in den unterschiedlichen Klassen verwendet werden.

Info

Bereits in VB 6 hat es die Möglichkeit gegeben, Interfaces zu nutzen. Jedoch war die Implementierung in VB 6 nicht wirklich intuitiv und komfortabel. Falls Sie das geärgert hat, vergessen Sie den Frust und freuen Sie sich auf Interfaces unter Visual Basic 10.

Wer sich mit den Interfaces in VB 6 bereits angefreundet hat, der wird dieses Sprachfeature jetzt in vollsten Zügen genießen.

Interfaces werden, wie gerade beschrieben, vor allem dann eingesetzt, wenn es darum geht, verschiedene Klassen für bestimmte Funktionalitäten gleich behandeln zu können oder mit sogenannten Providern zu arbeiten. Dabei wird innerhalb eines Providers bestimmte Funktionalität, wie zum Beispiel die Persistierung von Daten, implementiert. Sämtliche Provider implementieren dabei dieselben Schnittstellendefinitionen, jedoch unterschiedliche Logik. So kann der eine Provider Daten in eine XML-Datei speichern, während ein anderer Provider Daten in eine relationale Datenbank schreibt. Innerhalb Ihrer Applikation können Sie je nach Lust und Laune den gewünschten Provider wechseln (geht sogar nur mit Einträgen in die Konfigurationsdatei) und die gewünschte Funktionalität ausführen, ohne dabei Programmcode zu ändern.

Ein weiterer Anwendungsfall wäre, durch Interfaces die Beschreibung von Schnittstellen zu definieren, für welche die Logik noch nicht implementiert ist, um zum Beispiel bereits mit der Programmierung eines Clients beginnen zu können. Dies eignet sich besonders beim Prototyping.

Interfaces werden auch verwendet, um die Schnittstellen von Klassen bei verteilten Anwendungen auf den Clients bekanntzugeben, ohne die komplette Funktionalität auf den Clients ausliefern zu müssen.

Auch für den Fall, dass Sie Ihre .NET-Klassen in VB 6-Applikationen verwenden wollen, müssen Sie sämtliche öffentlichen Member Ihrer Klassen auch als Interfaces definieren, da dies der einzige Weg ist, die Schnittstellen unter COM (Komponentenobjektmodell) sichtbar zu machen und somit in VB 6 nutzen zu können. Mehr dazu in Kapitel 13.2 »COM-Interop«.

Nach dieser Einleitung will ich Ihnen in Listing 5.25 zeigen, wie Interfaces definiert und später innerhalb von Klassen implementiert werden.

Listing 5.25 Beispiel für eine Interface-Definition

```
Public Interface IPersistenz
    Property FileName() As String
    Sub Save()
    Sub Save(ByVal fileName As String)
End Interface
```

Hier definieren wir ein Interface mit dem Namen `IPersistenz`. Das Interface besitzt eine Eigenschaft `FileName` und eine überladene Methode `Save()`.

Innerhalb der Definition `Public Interface … End Interface` können Sie beliebig viele Methoden und Eigenschaften definieren.

Die Angabe eines Modifiers für den Gültigkeitsbereich ist nicht erlaubt, da in den implementierenden Klassen sämtliche Schnittstellen als `Public` definiert werden müssen. Auch statische Schnittstellen (mittels `Shared`) sind in einer Interface-Definition nicht erlaubt. Überladene Schnittstellendefinitionen sind, wie Sie in Listing 5.25 sehen, dagegen erlaubt.

Dieses, oder natürlich auch jedes andere, Interface können wir jetzt in jeder beliebigen Klasse implementieren. Eine Klasse kann dabei, im Gegensatz zur Vererbung, beliebig viele Interfaces definieren. Das Schlüsselwort zur Implementierung von Interfaces ist `Implements`.

Fügen Sie zu Ihren Klassen eine oder auch mehrere beliebige, durch Komma getrennte, `Implements`-Anweisungen hinzu. Wichtig ist hierbei die Stelle, an der die Implementierungen vorgenommen werden. `Implements`-Anweisungen müssen der `Inherits`-Anweisung (falls vorhanden) folgen und vor sämtlichem restlichen Programmcode der Klasse stehen.

Tipp

Wenn Sie die `Implements`-Anweisung getippt haben, verlassen Sie die Zeile nicht mit den Pfeil- oder sonstigen Tasten, sondern mit der Eingabetaste. Dadurch werden sämtliche Grundgerüste für die zu implementierenden Schnittstellen am Ende der Klassendefinition angelegt.

Ich habe der Klasse `Arbeiter` das Interface `IPersistenz` hinzugefügt und bin somit verpflichtet, sämtliche Schnittstellen, die in dem Interface definiert sind, auch in meiner Klasse zu implementieren.

Listing 5.26 zeigt die Implementierung des Grundgerüsts, das mir der Editor angelegt hat.

Listing 5.26 Implementierung eines Interface

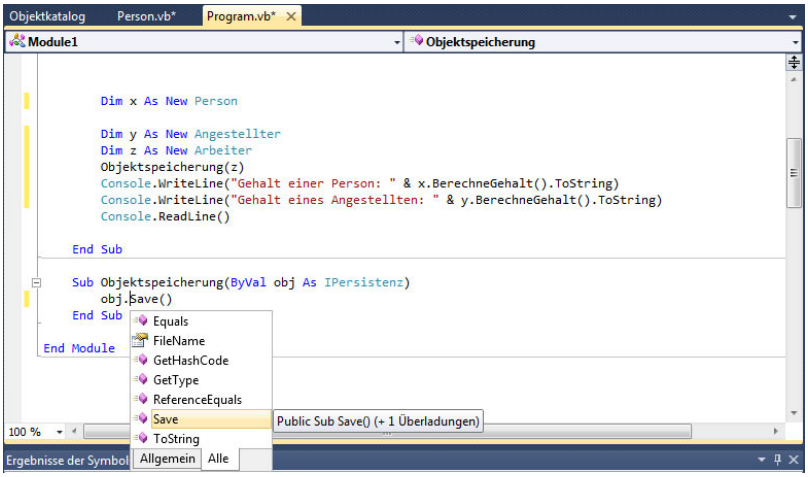
```
Public Class Arbeiter
    Inherits Person
    Implements IPersistenz
     …
    Public Property FileName() As String
      Implements IPersistenz.FileName
        Get

        End Get
        Set(ByVal value As String)

        End Set
    End Property

    Public Sub Save() Implements IPersistenz.Save

    End Sub

    Public Sub Save(ByVal fileName As String)
        Implements IPersistenz.Save

    End Sub
End Class
```

Sie müssen jetzt nur noch die entsprechende Logik hinterlegen, die in jeder Klasse, in der das Interface implementiert ist, unterschiedlich sein kann und in der Regel auch sein wird.

Die neu hinzugefügten Member können jetzt sehr einfach über eine Klasseninstanz aufgerufen werden. Sie werden keinen Unterschied zu den anderen implementierten Klassen-Membern sehen.

Ein sehr großer Vorteil von Interfaces liegt in der Tatsache begründet, dass sich Interfaces wie Datentypen verhalten können. Das heißt, Sie können bei Funktionen einen Parameter als Typ eines Interface definieren. Dadurch ist es möglich, jede beliebige Objektinstanz zu übergeben, die dieses Interface implementiert. Innerhalb dieser Routine stehen dann unabhängig vom übergebenen eigentlichen Objekttyp ausschließlich die Interface-Member zur Verfügung, wie Sie in Abbildung 5.105.10 sehen können.

Abbildung 5.10
Interface als Datentyp



Auch der bereits angesprochene Providergedanke ist mit Interfaces sehr einfach zu realisieren. Sie wissen unter Umständen erst zur Laufzeit, welche Implementierung für eine bestimmte Aufgabe verwendet werden soll. Jetzt können Sie eine Variable vom Typ eines Interface definieren und müssen dann nur bei der Instanziierung den gewünschten Typ angeben. Das spart in der Regel sehr viel Programmcode und macht Ihre Applikationen bei weitem flexibler und wartbarer. Ein kleines Beispiel dazu sehen Sie in Listing 5.27.

Listing 5.27
Interfaces zur Nutzung unterschiedlicher Provider

```
Sub SerializeMe(ByVal soap As Boolean, _
    ByVal fileName As String)
    Dim Serializer As _
        System.Runtime.Serialization.IFormatter
    If soap Then
       Serializer = New
System.Runtime.Serialization.Formatters.Soap.SoapFormatter()
    Else
       Serializer = New
System.Runtime.Serialization.Formatters.Binary.BinaryFormatter
```

```
    End If
    Dim fs As New System.IO.FileStream(
       fileName, IO.FileMode.Create)
    Serializer.Serialize(fs, Me)
End Sub
```

Listing 5.27 (Forts.)
Interfaces zur Nutzung unterschiedlicher Provider

In Listing 5.27 wird eine Variable `Serializer` definiert vom Typ des Interface `IFormatter`. In Abhängigkeit der übergebenen booleschen Variablen `soap` wird diese Variable vom Typ eines `BinaryFormatter` oder `SoapFormatter` instanziiert. Danach können alle Eigenschaften und Methoden dieses Interface auf diese Objektvariable angewandt werden.

5.3.2 Abstrakte Basisklassen

Ähnlich wie Interfaces setzen abstrakte Basisklassen auf Typenkompatibilität durch fest definierte Schnittstellen, die mit unterschiedlicher Funktionalität hinterlegt sind. Im Gegensatz zu den Interfaces jedoch können bei den abstrakten Basisklassen bereits bestimmte Schnittstellen mit einer Implementierung versehen sein. Wir haben somit eine Mischung aus vollständig implementierten Methoden und so genannten virtuellen Methoden, die keine Implementierung besitzen. Eine abstrakte Basisklasse ist jedoch eine besondere Form einer Klasse, denn sie kann nicht instanziiert werden. Von einer abstrakten Basisklasse kann somit nur abgeleitet und alle virtuellen Methoden müssen dann in der abgeleiteten Klasse implementiert werden.

Der wesentliche Unterschied zu Interfaces ist, dass abstrakte Basisklassen über Vererbung funktionieren und somit eine Klasse nicht beliebig viele andere abstrakte Basisklassen implementieren kann, was bei den Interfaces ja möglich ist.

Wenn Sie sehr viele Objekte mit vielen Gemeinsamkeiten haben, die sich jedoch in Detailsachen unterscheiden, sollten Sie sich überlegen, ob Sie abstrakte Basisklassen nutzen wollen, anstatt in den einzelnen Klassen jeweils mehrfach die gemeinsame Logik zu implementieren.

Zum Beispiel gibt es bei einer Bank sehr viele unterschiedliche Konten, ein Girokonto, ein Sparkonto, ein Gehaltskonto und noch viele mehr. Alle Konten haben viele Gemeinsamkeiten (Kontonummer, Inhaber, Kundenbetreuer, Einzahlen, Auszahlen etc.), aber auch Unterschiede (Gebühren berechnen, Zinsen berechnen, Dispo). Innerhalb einer Basisklasse `Konto` könnte man nun die komplette Implementierung der gemeinsamen Funktionalitäten vornehmen und die unterschiedlichen Funktionalitäten würde man als virtuelle Methoden implementieren, die dann in den jeweils speziellen Klassen, abgeleitet von der abstrakten Basisklasse `Konto`, mit Logik befüllt werden müssen.

Eine abstrakte Klasse wird bei der Definition mit dem Schlüsselwort `MustInherit` gekennzeichnet. Die virtuellen Member werden zusätzlich mit dem Schlüsselwort `MustOverride` versehen. Sobald eine Methode mit `MustOverride` gekennzeichnet ist, muss auch die Klasse mit `MustInherit` versehen werden, ansonsten würde der Compiler einen Fehler melden.

Ein Beispiel für eine abstrakte Basisklasse sehen Sie in Listing 5.28.

Listing 5.28 Beispiel für die Definition einer abstrakten Basisklasse

```
Public MustInherit Class Konto
    Private mKontoStand As Double
    Public Property Kontostand() As Double
       Get
            Return mKontostand
       End Get
       Set(ByVal value As Double)
           mKontostand = value
       End Set
    End Property

    Public Sub Einzahlen(ByVal betrag As Double)
        Kontostand += betrag
    End Sub

    Public MustOverride Sub GebuehrBerechnen()
    Public MustOverride Sub ZinsBerechnen()
End Class
```

In diesem Beispiel haben wir zwei virtuelle Methoden `GebuehrBerechnen()` und `ZinsBerechnen()` implementiert. Außerdem haben wir eine Property `Kontostand` und eine Methode `Einzahlen()` innerhalb der Klasse, deren Logik schon vollständig vorhanden ist. Dadurch, dass wir in der Klasse zwei virtuelle Methoden besitzen, muss die Klasse zwingend als abstrakte Basisklasse definiert werden.

Listing 5.29 zeigt die Ableitung von dieser abstrakten Basisklasse `Konto`.

Listing 5.29 Beispiel für eine von einer abstrakten Basisklasse abgeleiteten Klasse

```
Public Class GiroKonto
    Inherits Konto

    Public Overrides Sub GebuehrBerechnen()
        'Implementierung der Funktionalität
    End Sub

    Public Overrides Sub ZinsBerechnen()
    'Implementierung der Funktionalität
    End Sub
End Class
```

In einer neuen Klasse, die von einer abstrakten Basisklasse erbt, müssen alle virtuellen Methoden der Basisklasse implementiert werden. Dabei verwenden Sie, wie bei Vererbung üblich, das Schlüsselwort `Overrides`.

Durch abstrakte Basisklassen ist es möglich, ein komplexeres Klassen-Framework übersichtlich und wartbar zu gestalten.

5.3.3 Polymorphie

Polymorphie kommt aus dem Griechischen und bedeutet Vielgestaltigkeit. Unter Polymorphie versteht man in der Programmierung, durch einen Aufruf einer Methode innerhalb von typkompatiblen Klassen unterschiedliche Funktionalitäten erhalten zu können.

Dies nennt man auch **virtuelle Programmierung**.

Nehmen wir an, wir haben ein Array von verschiedenen Kontoobjekten. Alle besitzen dieselben Schnittstellen, nur die Implementierung von bestimmten Methoden ist durch Überschreiben in den Klassen unterschiedlich. Wenn man jetzt eine dieser Methoden für jedes Objekt in diesem Array aufrufen will, dann braucht man dazu eine Objektvariable vom Typ der Basisklasse, wenn man auf `Select Case` mit Typabfragen verzichten will. Obwohl die Objektvariable vom Typ der Basisklasse definiert ist, wird die Implementierung so ausgeführt, wie sie in der jeweiligen speziellen Kontoklasse hinterlegt wurde. Wurde in einer vererbten Klasse die Methode nicht überschrieben, so wird die Implementierung der Basisklasse genommen. Und genau diesen Vorgang nennt man Polymorphie, da die Ausführung vielgestaltig sein kann.

In Listing 5.30 und Listing 5.31 will ich Polymorphie an einem kleinen Beispiel aufzeigen.

Listing 5.30
Klassengerüst für Polymorphie

```
Public Class BaseClass
 Public Overridable Function DoSomething() As String

 Return Me.GetType.ToString & _
 " Ich bin die allgemeine Klassenimplementierung"

 End Function
End Class

Public Class NormalClass
    Inherits BaseClass
End Class

Public Class SpecialClass
 Inherits BaseClass
 Public Overrides Function DoSomething() As String
   Return Me.GetType.ToString & _
  " Ich bin die spezielle Klassenimplementierung"
 End Function
End Class

Public Class SuperSpecialClass
 Inherits BaseClass
 Public Overrides Function DoSomething() As String
  Return Me.GetType.ToString & _
  " Ich bin die superspezielle Klassenimplementierung"
 End Function
End Class
```

Innerhalb von Listing 5.30 sehen Sie die Definition einer Klasse, die eine überschreibbare Methode `DoSomething()` enthält. Außerdem sind drei weitere Klassen definiert, die sich von `BaseClass` ableiten, wobei zwei davon die entsprechende Methode überschreiben. In der überschriebenen Methode wird dabei ein String zurückgeben, der den Typ der Klasse darstellt und einen mehr oder wenig sinnvollen Text.

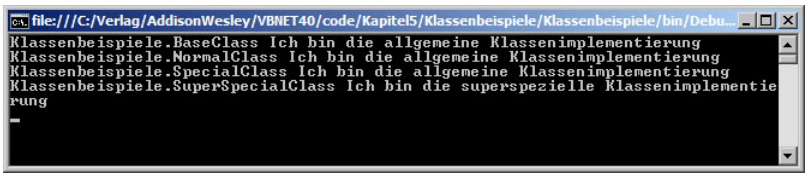
In Listing 5.31 wird nun eine Instanz eines jeden Typs erstellt und alle Objekte werden in einem Array zusammengefasst. Über eine Schleife wird

schließlich die Methode `DoSomething()` einmal für jedes Objekt aufgerufen. Die Bildschirmausgabe sehen wir in Abbildung 5.11.

Listing 5.31
Polymorphie darstellen

```
Dim b As New BaseClass
Dim bn As New NormalClass
Dim bs As New SpecialClass
Dim bss As New SuperSpecialClass
Dim arrB() As BaseClass = {b, bn, bs, bss}

For Each ba As BaseClass In arrB
    Console.WriteLine(ba.DoSomething)
Next
```

Abbildung 5.11
Ausgabe des Beispiels aus Listing 5.31



Wir erhalten, dank Polymorphie, vier unterschiedliche Ausgaben, da jedes Mal, wie wir auch an der Ausgabe des Klassennamens erkennen können, die Implementierung der speziellen Klasse verwendet wurde.

Es besteht die Möglichkeit, mittels des Schlüsselworts `Shadows` die Polymorphie abzuschalten. Innerhalb der Klasse würde die Methode dann nicht überschrieben werden, sondern lediglich **verschattet oder überdeckt**. Das bedeutet, dass, falls die Methode im Kontext der Basisklasse aufgerufen wird, Polymorphie nicht greift und die Basisklassenimplementierung verwendet wird, ansonsten wird die tatsächliche Implementierung ausgeführt.

Dazu habe ich in der Klasse `SpecialClass` die Definition der Funktion `DoSomething()` folgendermaßen geändert:

```
Public Shadows Function DoSomething() As String
```

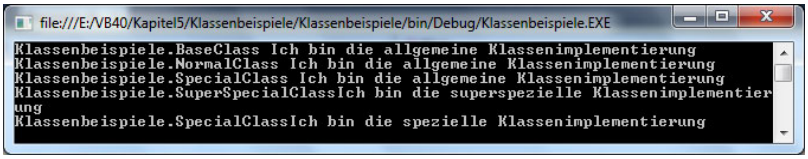
Und den Code aus Listing 5.31 um folgende Zeile ergänzt:

```
Console.WriteLine(bs.DoSomething)
```

Dadurch wird zusätzlich die Methode `DoSomething()` im Kontext der Klasse `SpecialClass` ausgeführt.

In Abbildung 5.12 sehen wir die Bildschirmausgabe nach den kleinen Änderungen.

Abbildung 5.12
Ausgabe mit Shadows



Wie wir sehen, wird innerhalb der Schleife für den Typ `SpecialClass` die Basisklassenimplementierung verwendet, da wir bei `SpecialClass` das Schlüssel-

wort Shadows anstatt Overrides verwendet haben. Am Ende (also nach der Schleife) haben wir wieder die Ausgabe, so wie in der Klasse implementiert.

5.4 Strukturen

Strukturen wurden bereits in VB.NET 1.0 als eine Art Nachfolger der benutzerdefinierten Datentypen (Types) eingeführt. Konnte man mittels der Type-Definition lediglich mehrere Eigenschaften, mit unterschiedlichem Datentyp, zu einem neuen Typ zusammenfassen, so bieten Strukturen unter .NET bei weitem mehr.

Auf den ersten Blick ähneln Strukturen den Klassen, denn Strukturen unterstützen Properties, Methoden, Events und sogar Konstruktoren. Betrachtet man Strukturen aber etwas genauer, so wird man sehr schnell feststellen, dass sich Strukturen ganz deutlich von Klassen unterscheiden und gewissen Einschränkungen unterliegen.

Der größte Unterschied liegt in der Tatsache begründet, dass es sich bei Strukturen um Wertetypen handelt und nicht um Referenztypen, wie es bei Klassen der Fall ist. Das bedeutet, dass Strukturen im **Stack** des Speichers liegen und auf diese somit auch sehr schnell zugegriffen werden kann. Eine Zuweisung mit dem =-Operator ist ein Kopieren einer Struktur und nicht das Zuweisen einer Speicheradresse an eine Objektvariable, wie es bei den Klassen der Fall ist. Dadurch, dass eine Struktur im Stack des Speichers liegt, wird ihr Speicher auch nicht durch den Garbage Collector freigegeben.

Eine wesentliche Einschränkung ist, dass Strukturen nicht vererbt werden können, **Vererbung** ist nur für Klassen möglich (siehe Abschnitt 2 Vererbung). Jedoch ist das Implementieren von Interfaces (siehe Abschnitt 1 Interfaces) auch bei Strukturen möglich.

Die Verwendung des Schlüsselworts WithEvents zum einfachen Definieren einer Ereignisroutine ist bei Strukturen auch nicht zulässig. Um einer Struktur eine Ereignisroutine zuzuweisen, muss dies dynamisch per Programmcode mittels der AddHandler-Anweisung geschehen.

Die Definition einer Struktur steht zwischen den beiden Schlüsselwörtern Structure Name_der_Struktur und End Structure. Innerhalb dieses Blocks können nun – ähnlich wie bei den Klassen – Methoden, Properties und Events definiert werden.

Innerhalb von Strukturen oder Klassen können wiederum andere Strukturen definiert werden.

Listing 5.32 zeigt ein Beispiel für eine Struktur:

```
Public Structure structPerson
    Public Event AlterChanged (ByVal sender As Object, _
        ByVal e As EventArgs)
```

Listing 5.32
Beispiel für eine Struktur

Listing 5.32 (Forts.)
Beispiel für eine Struktur

```
    Sub New(ByVal personalNummer As String)
        Me.PersonalNummer = personalNummer
    End Sub

    Private mPersonalNummer As String
    Property PersonalNummer() As String
        Get
             Return mPersonalNummer
        End Get
        Set(ByVal value As String)
            mPersonalNummer = value
        End Set
    End Property

    Private mAlter As Short
    Public Property Alter() As Short
        Get
            Return mAlter
        End Get
        Private Set(ByVal value As Short)
            If value >= 0 And value < 150 Then
                mAlter = value
                 RaiseEvent AlterChanged (Me, New EventArgs())
             Else
                 Throw New Exception("Ungültiges Alter")
             End If
        End Set
    End Property
End Structure
```

Das Beispiel habe ich ganz bewusst so gewählt wie das vorhergehende Klassenbeispiel in diesem Kapitel, um zu zeigen, dass Strukturen ähnlich wie Klassen programmiert werden.

Um eine Struktur nun zu nutzen, kann sie sowohl mit dem Schlüsselwort `New` wie auch ohne erzeugt werden.

Alle drei folgenden Varianten wären gültig, um eine Struktur zu erzeugen:

```
Dim uPerson As structPerson
```

oder:

```
Dim uPerson As New structPerson("2606-C80-162")
```

oder:

```
Dim uPerson As New structPerson()
```

Wobei bei der ersten Variante die Variable `uPerson` vor der ersten Verwendung noch zugewiesen werden muss.

Überraschend ist etwas, dass die letzte aufgeführte Variante mit `New` und ohne Konstruktorparameter funktioniert, dies wäre bei Klassen nicht möglich gewesen. Aber wie bereits erwähnt, wird die Struktur nicht im Heap instanziiert, sondern im Stack angelegt. Deswegen spricht man bei Strukturen auch nicht von Instanziierung.

Strukturen sind in manchen Fällen durchaus eine Alternative zu Klassen, jedoch fehlt mit Vererbung ein ganz wesentlicher Bestandteil objektorientierter

Programmierung, um sich einen übersichtlichen, wartbaren und nicht redundanten Code zu erzeugen. Nicht zuletzt aufgrund des schnelleren Speicherzugriffs bieten sie aber manchmal durchaus auch Vorteile gegenüber Klassendefinitionen.

5.5 Attribute

Attribute in .NET unterstützen das **deklarative Programmiermodell**. Das bedeutet, dass über diese Attribute Informationen an eine Assembly, an Klassen, an Methoden und so weiter angehängt werden können, die dann von der Laufzeitumgebung während der Ausführung des Programms ausgewertet werden. Ein bestimmtes Verhalten wird somit deklariert und nicht programmiert.

Achtung

Bitte verwechseln Sie das Wort Attribute nicht mit der Bedeutung in der objektorientierten Theorie. Der Begriff Attribute aus der objektorientierten Welt entspricht unter .NET dem Property-Begriff.

Die Attributinformationen werden dabei mit den Metadaten des Elements gespeichert und können über **Reflections** zur Laufzeit ausgewertet werden. Dabei haben Sie auch die Möglichkeit, eigene Attribute zu definieren.

Ein Attribut ist eine Klasse, die von `System.Attribute` abgeleitet wird. Jedes Attribut, das Sie in .NET zur Verfügung haben, basiert auf einer Klasse, deren Name sich aus dem Attributnamen und dem Wort `Attribute` zusammensetzt.

Jedes Projekt besitzt eine *AssemblyInfo*-Datei, die ausschließlich aus Attributen besteht, die für die gesamte Assembly gelten. Innerhalb dieser Datei können Sie sich einen Überblick verschaffen, wie Attribute aussehen und welche Informationen hier an eine Assembly gehängt werden.

Listing 5.33 zeigt einen Ausschnitt aus einer *AssemblyInfo.vb*-Datei:

Listing 5.33
Assembly-Info

```
<Assembly: AssemblyTitle("Klassenbeispiele")>
<Assembly: AssemblyDescription("")>
<Assembly: AssemblyCompany _
    ("Prime Time Software")>
<Assembly: AssemblyProduct("Klassenbeispiele")>
<Assembly: AssemblyCopyright _
    ("Copyright © Prime Time Software 2010")>
<Assembly: AssemblyTrademark("")>
```

Attribute werden implementiert, indem Sie vor der entsprechenden Klasse oder Methode das Attribut innerhalb von spitzen Klammern schreiben. Die Parameter, die in Klammern angegeben werden, werden dabei an den Konstruktor der Attributklasse übergeben.

Sie werden mit Attributen sehr oft in Berührung kommen und sollten sich deswegen auch nicht scheuen, in der Hilfe einfach nach dem entsprechenden Eintrag für dieses Attribut zu suchen. Dabei wird Ihnen vielleicht viel klarer, warum dieses Attribut an dieser Stelle stehen soll und was tatsächlich im Hintergrund geschieht.

5.6 Delegates und Events

Delegates sind in .NET neu eingeführt worden. Dabei handelt es sich um typsichere Funktionszeiger, die Funktionen mit identischer Signatur über deren Adresse aufrufen können. Im Gegensatz zu Pointer kümmert sich die .NET-Laufzeitumgebung um die richtige Adressierung, falls sich Speicheradressen verschieben, zum Beispiel nach einem Lauf des Garbage Collector. Das gesamte Event-System innerhalb des .NET Framework basiert auf Delegates.

5.6.1 Delegates

Mittels eines Delegate können Sie durch einen einzigen Aufruf mehrere Funktionen in unterschiedlichen Klassen gleichzeitig aufrufen. Voraussetzung dafür ist nur, dass alle Funktionen eine einheitliche Signatur besitzen und die Funktionsadressen dem Delegate hinzugefügt wurden. Die Funktionsnamen können dabei völlig unterschiedlich sein. Somit können Sie bei der Instanziierung von Objekten eine bestimmte Funktion dem Delegate hinzufügen und ab dem Zeitpunkt, zu dem Sie das Objekt nicht mehr benötigen, die Funktionsadresse wieder entfernen.

Ein Delegate wird folgendermaßen definiert:

```
Delegate Sub Delegatename(Parameter)
```

Ein Delegate muss vor der Verwendung, genauso wie eine Klasse, instanziiert werden. Dies geschieht mit folgender Syntax:

```
Dim myOwnDelegate As New Delegatename AddressOf MyProzToCall)
```

Achtung

Bitte beachten Sie, dass Address im Englischen mit zwei d geschrieben wird. Ein sehr beliebter Fehler!

Danach kann das Delegate ganz einfach wie folgt aufgerufen werden:

```
myOwnDelegate(Parameter)
```

Bereits bei der Definition eines Delegate wird durch die Angabe der Parameter die Signatur bestimmt. Diesem Delegate können danach nur noch Funktionsadressen mit identischer Signatur zugewiesen werden.

Um einem Delegate weitere Funktionsadressen hinzuzufügen, können Sie die statische `Combine()`-Methode der Delegate-Klasse aufrufen.

Eine kleine Konsolenapplikation, die in Listing 5.34 dargestellt ist, soll die Verwendung von Delegates illustrieren.

Listing 5.34 Beispiel für ein Delegate

```
Module Module1

    Delegate Sub myDelegate(ByVal test As String)

    Sub Main()
        Dim myOwnDelegate As New myDelegate(AddressOf Proz1)
        myOwnDelegate("test")
        Dim myOwnDelegate2 As New myDelegate(AddressOf Proz2)
```

Listing 5.34 (Forts.)
Beispiel für ein Delegate

```
        myOwnDelegate = _
            CType(System.Delegate.Combine _
                (myDelegate, myDelegate2), myDelegate)
        myOwnDelegate("Test mit zwei Delegates")
        Console.ReadLine()
    End Sub

    Sub Proz1(ByVal test As String)
        Console.WriteLine("Proz1 sagt: " & test)
    End Sub
    Sub Proz2(ByVal test As String)
        Console.WriteLine("Proz2 sagt: " & test)
    End Sub
End Module
```

In diesem Beispiel definieren wir ein Delegate `mydelegate`, das Prozeduren aufrufen kann, die einen Stringparameter erwarten. Innerhalb des Moduls gibt es zwei Methoden `Proz1()` und `Proz2()`, die dieser Signatur entsprechen. In der `Main()`-Routine schließlich instanziieren wir zwei Delegate-Objekte `myOwnDelegate` und `myOwnDelegate2`. Mit der Methode `Combine()` fügen wir `myOwnDelegate2` dem ersten Delegate-Objekt `myOwnDelegate` hinzu. Beachten Sie bitte, dass an dieser Stelle auch eine Typumwandlung von `Delegate` auf `myDelegate` stattfindet, da die Methode `Combine()` ein Objekt vom Typ `System.Delegate` zurückgibt.

In Abbildung 5.13 ist die Ausgabe dieser kleinen Applikation dargestellt. Zuerst zeigt das Delegate beim Aufruf mit dem Parameter `"test"` nur auf `Proz1`. Beim zweiten Aufruf zeigt es bereits auf beide Methoden, so dass wir als Ausgabe insgesamt drei Zeilen bekommen.

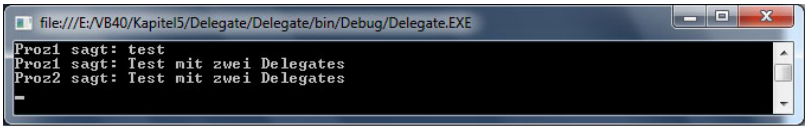


Abbildung 5.13
Ausgabe des Delegate-Beispiels

Ein weiteres Einsatzgebiet von Delegates sind **Callback**-Funktionen. Dabei übergeben Sie an eine Funktion ein Delegate, das dann von dieser entsprechenden Funktion wieder aufgerufen wird. Solche Beispiele werden Sie vor allem bei asynchronen Funktionsaufrufen sehen. Außer den Funktionsparametern übergeben Sie der asynchronen Funktion noch eine Adresse einer Funktion als Delegate, die nach vollständiger Abarbeitung zurückgerufen wird, um das Funktionsergebnis zurückzugeben.

5.6.2 Events basierend auf Delegates

Wie in der Einführung bereits erwähnt, basiert das gesamte .NET-Event-System auf Delegates. Dies werden Sie spätestens dann sehen, wenn Sie sich den Code Ihrer Ereignisroutinen in einer Windows-Applikation genauer anschauen.

Sie werden dabei feststellen, dass hinter jeder Methodensignatur eine `Handles`-Anweisung steht. Über diese `Handles`-Anweisung wird eine Verbindung der Methode, dem **Ereignishandler**, und dem gewünschten Event hergestellt. Der Name der Methode ist dabei frei wählbar, er muss somit nicht mehr in der in VB 6 ursprünglichen Syntax `ObjektName_Ereignisname` geschrieben werden, auch wenn der Ereignishandler standardmäßig mit diesem Namen vorgeschlagen wird.

Tipp

Probieren Sie doch, die `Form_Load()`-Methode auf einen beliebigen Namen umzubenennen. Solange Sie nichts an der `Handles`-Anweisung ändern, wird Ihr Programm weiterhin korrekt ausgeführt werden.

Sehr oft hat man in einem Programm auch die Anforderung, dass ein und dieselbe Funktionalität über verschiedene Wege aufgerufen werden kann (Klicken auf einen Button, Auswahl in einem Menü oder Kontextmenü etc.). Durch die `Handles`-Anweisung ist es möglich, all diese Ereignisse auf ein und denselben Ereignishandler zu richten, indem Sie alle gewünschten Ereignisse mit Kommata getrennt hinter dem Schlüsselwort `Handles` angeben.

```
Private Sub btnBerechne_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles btnBerechne.Click, _
    BerechneToolStripMenuItem.Click, _
    BerechneToolStripMenuItem1.Click
```

Der Ereignishandler `btnBerechne_Click()` reagiert jetzt auf alle drei Click-Ereignisse der verschiedenen Objekte.

Achtung

Sie müssen hierbei nur aufpassen, dass sämtliche Ereignisse, die Sie in der Handles-Anweisung angeben, auch eine identische Signatur besitzen. Ein `MouseMove`-Ereignis würde im obigen Beispiel vom Compiler als fehlerhaft angemerkt werden.

Mittels der `Handles`-Anweisung werden die Ereignishandler fest mit den gewünschten Events verdrahtet. Es gibt aber auch die Möglichkeit einer dynamischen Zuordnung, die uns sehr viel mehr Möglichkeiten und Freiheiten als vor der .NET-Zeit bietet.

Diese dynamische Zuordnung benötigen Sie immer, wenn Sie per Programmcode Steuerelemente zu Ihrem Formular hinzufügen, um auch auf Ereignisse der neuen Elemente reagieren zu können.

Manchmal wünscht man sich auch, dass bestimmte Ereignisse noch nicht feuern, wenn sie bei einer Vorinitialisierung eines Formulars aufgerufen werden (`TextBox_Changed`, `ComboBox_SelectedIndexChanged` etc.). Durch diese dynamische Zuordnung kann der Event auch erst nach Abarbeitung der `Form_Load()`-Methode einem Ereignishandler zugewiesen werden.

Dafür steht Ihnen die Anweisung `AddHandler` beziehungsweise zum Auflösen der Zuordnung die Anweisung `RemoveHandler` zur Verfügung.

Listing 5.35 zeigt die Funktionsweise der beiden Anweisungen.

Listing 5.35 Funktionsweise von Add- und RemoveHandler

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.
EventArgs) Handles MyBase.Load

    ComboBox1.Selectedindex = 0
    AddHandler ComboBox1.SelectedIndexChanged,
        AddressOf Listenauswahl_Geaendert
End Sub

Private Sub Listenauswahl_Geaendert(ByVal sender As Object,
    ByVal e As System.EventArgs)

        MessageBox.Show(ComboBox1.SelectedItem.ToString())
        RemoveHandler ComboBox1.SelectedIndexChanged,
            AddressOf Listenauswahl_Geaendert
End Sub
```

In diesem Beispiel wird mittels `AddHandler` erst am Ende von `Form_Load()` dem Ereignis `SelectedIndexChanged` der `ComboBox1` als Ereignishandler die Methode `Listenausahl_Geaendert()` hinzugefügt. Wichtig hierbei ist, dass `Listenauswahl_Geaendert()` eine identische Signatur, wie das auf dieses Ereignis basierende Delegate, besitzt.

Das bedeutet, dass durch die Programmzeile `ComboBox1.Selectedindex = 0` keine Meldung am Bildschirm ausgegeben wird. Erst wenn Sie danach den Index ändern, erscheint die Meldung, aber auch nur ein einziges Mal. Denn mittels `RemoveHandler` wird die Funktionsadresse von `Listenauswahl_Geaendert()` wieder aus dem Delegate für das Ereignis `ComboBox1_SelectedindexChanged` entfernt.

Diese beiden Anweisungen bieten uns vollständig neue Möglichkeiten, die wir mit VB 6 nicht hatten, und deshalb sollten Sie diese immer in Ihrem Gedächtnis behalten.

5.6.3 Relaxed Delegates

Bei allen Ereignishandlern sowohl bei Webanwendungen (ASP.NET) wie auch bei Windows-Anwendungen (Windows Forms) werden in der Regel zwei Parameter übergeben. Ein `Sender` vom Typ `Object`, über den die Referenz auf das auslösende Ereignis herzustellen ist, und ein Objekt vom Typ `EventArgs` oder eine von `EventArgs` abgeleiteten Klasse, in der zusätzliche Informationen zu dem Ereignis stehen. In den meisten Fällen werden diese Parameter nicht ausgewertet, sie müssen jedoch angegeben werden, da ansonsten die Schnittstelle verletzt wird.

Dank der Relaxed Delegates ist dies seit Visual Basic 9 nicht mehr nötig.

Sie können jetzt auch parameterlose Methoden als Ereignishandler verwenden oder auch Methoden, die nur in einem Teil der Signatur entsprechen, solange sie eindeutig zuzuordnen sind.

Nehmen wir als Beispiel die normale Ereignissignatur für das Ereignis `Load` eines Formulars.

```
Private Sub Form1_Load(ByVal sender As Object,
       ByVal e As EventArgs) Handles MyBase.Load
```

Dank Relaxed Delegates kann der Ereignishandler jetzt auch wie folgt definiert werden:

```
Private Sub Form1_Load() Handles MyBase.Load
```

Oder auch:

```
Private Sub Form1_Load(ByVal sender As Object,
          ByVal e As Object) Handles MyBase.Load
```

Aber nicht:

```
Private Sub Form1_Load(ByVal sender As Object)
        Handles MyBase.Load
```

Bei dieser Überladung kann der Compiler nicht entscheiden, ob der Parameter für den Auslöser des Events steht oder für die Eventargumente, da `EventArgs` ja von `Object` erbt.

Um die Sache zu verdeutlichen, will ich das Beispiel noch mal mit einem spezielleren Ereignishandler zeigen.

Statt

```
Private Sub Form1_MouseDown(ByVal sender As System.Object,
   ByVal e As MouseEventArgs) Handles MyBase.MouseDown
```

können Sie auch

```
Private Sub Form1_MouseDown() Handles MyBase.MouseDown
```

oder

```
Private Sub Form1_MouseDown(ByVal sender As System.Object,
   ByVal e As EventArgs) Handles MyBase.MouseDown
```

schreiben.

Aber wiederum nicht:

```
Private Sub Form1_MouseDown(ByVal e As MouseEventArgs)
    Handles MyBase.MouseDown
```

Auch hier kann der Compiler keine eindeutige Zuordnung durchführen.

Bei einem Blick in den IL-Code werden Sie sehen (Abbildung 5.14 und Abbildung 5.15), dass der Compiler eine eigene Methode mit einer passenden Signatur anlegt, um die Methode mit verkürzter Signatur intern aufzurufen.

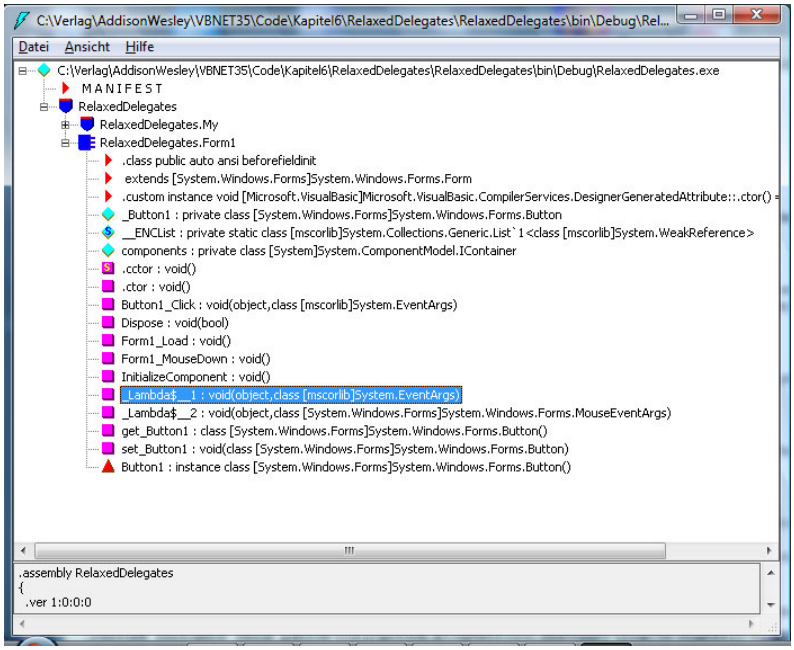


Abbildung 5.14
IL-Code zur Darstellung der Relaxed Delegates

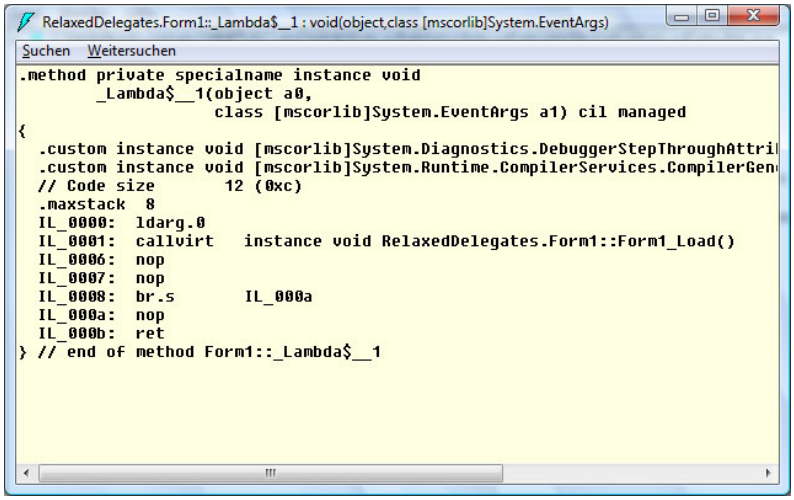


Abbildung 5.15
Wrapper-Methode zum Aufruf von Relaxed Delegate Ereignishandler in IL-Code

Achtung

Die C#-Entwickler sind an dieser Stelle nicht ganz so entspannt wie die VB-Entwickler, denn Relaxed Delegates sind nur in Visual Basic implementiert.

5.7 Generics

Sollten Sie in der Vergangenheit bereits viel mit **Collection-Objekten** gearbeitet haben, ist doch immer und immer wieder dasselbe Problem aufgetreten. Typensicherheit der Objekte innerhalb der Collection war nicht gegeben, da eine Collection standardmäßig eine Auflistung von Objekten, egal welchen Typs, ist.

Also machte man sich zumeist selbst an die Arbeit und hat mit mehr oder weniger Aufwand eigene typsichere Auflistungsobjekte geschrieben. Der Aufwand bei großen Objektmodellen war jedoch gewaltig, musste man doch für jeden unterschiedlichen Typ eine eigene Collection-Klasse schreiben.

Als zweites Problem kommt hinzu, dass bei der Verwendung von eigens entwickelten typsicheren Klassen intern immer zeitaufwändige Typumwandlungen stattfinden, um das Objekt auch mit dem korrekten Datentyp zurückgeben zu können. Mit diesen Typumwandlungen mussten auch alle Entwickler kämpfen, die sich den Aufwand nicht machen wollten, typsichere Collections zu entwickeln. Jeder Zugriff auf die `Items`-Eigenschaft musste mit `CType()` versehen werden, um auch den gewünschten Typ zu erhalten.

Mit .NET 2.0 hat Microsoft ein neues Konzept eingeführt, das in etwa mit den *Templates* in C vergleichbar ist und alle oben beschriebenen Probleme löst. Generische Auflistungsobjekte, **Generics**, wurden in einem eigenen Namensraum `System.Collections.Generic` bereitgestellt. Die bisher eingeführten Collection-Objekte sind weiterhin völlig abwärtskompatibel in `System.Collections` vorhanden.

Was bieten uns nun aber diese Generic-Klassen? Die gute Nachricht ist: Sie bieten all das, was wir uns seit Jahren wünschen. Beim Konstruktoraufruf müssen wir lediglich den Typ angeben, für den die Auflistung verwendet werden soll, bei Schlüssel-Werte-Paar-Auflistungen können wir auch den Typ des Schlüssels angeben und schon ist diese Auflistung typensicher. Es können nur noch Objekte von diesem speziellen Typ hinzugefügt werden, und auch die Rückgabewerte aus diesem speziellen Auflistungsobjekt sind nicht mehr vom Typ `Object`, sondern genau von diesem speziell angegebenen Typ. Dabei finden auch intern keine zeitaufwändigen Typumwandlungen statt, denn diese Klassen werden generisch erzeugt und sind vollkommen compilerunterstützt. Sogar **IntelliSense** funktioniert an den Stellen, wo Objekte dieses Typs zurückgegeben werden.

Info

Eine Collection mit einer Schlüssel-Werte-Paar-Beziehung ist eine Auflistung, in der zu jedem Objekt zusätzlich ein eindeutiger Schlüsselwert gespeichert wird. Über diesen Schlüssel kann man dann sehr schnell auf das gewünschte Objekt zugreifen, ohne die Auflistung sequenziell durchlaufen zu müssen.

Innerhalb des Namensraums `System.Collections.Generic` gibt es eine ganze Reihe von unterschiedlichen generischen Auflistungsobjekten, deren wichtigste ich in Tabelle 5.2 kurz aufliste.

Klasse	Beschreibung
`Dictionary`	Stellt ein Auflistungsobjekt mit einer Schlüssel-Werte-Paar-Beziehung dar.
`LinkedList`	Stellt eine doppelt verknüpfte Liste dar.
`List`	Stellt ein Auflistungsobjekt dar, über das über den Index auf die Objekte zugegriffen werden kann.
`Queue`	Stellt ein First-In-First-Out(FIFO)-Auflistungsobjekt dar.
`SortedDictionary`	Stellt ein Auflistungsobjekt mit einer Schlüssel-Werte-Paar-Beziehung dar, die nach dem Schlüsselbegriff sortiert ist.
`SortedList`	Stellt ein Schlüssel-Werte-Paar-Auflistungsobjekt dar, das aufgrund einer zugeordneten `IComparer`-Implementierung sortiert wird.
`Stack`	Stellt ein Last-In-First-Out(LIFO)-Auflistungsobjekt dar.

Tabelle 5.2
Klassen im NamespaceSystem.Collections.Generic

Wichtig dabei ist auch die Betrachtung der Interfaces, welche die unterschiedlichen Generic-Klassen implementieren. Diese will ich in Tabelle 5.3 kurz darstellen.

Interface	Beschreibung
`ICollection`	Definiert folgende Methoden und Eigenschaften für generische Auflistungen: ■ Add() ■ Clear() ■ Contains() ■ CopyTo() ■ Count (schreibgeschützte Eigenschaft) ■ IsReadOnly (schreibgeschützte Eigenschaft) ■ Remove() Implementiert außerdem das Interface `IEnumerable`.
`IComparer`	Definiert eine Methode `Compare()` zum Vergleichen zweier Objekte.
`IDictionary`	Definiert folgende Methoden und Eigenschaften für generische Auflistungen mit Schlüssel-Werte-Paar-Beziehungen: ■ Add() (zusätzliche Überladung) ■ ContainsKey() ■ Item (Eigenschaft) ■ Keys (Eigenschaft) ■ Remove() (zusätzliche Überladung) ■ TryGetValue() ■ Values (schreibgeschützte Eigenschaft) Implementiert außerdem die Interfaces `ICollection` und `IEnumerable`.

Tabelle 5.3
Interfacedefinitionen im Namespace System.Collections.Generic

Tabelle 5.3 (Forts.) Interfacedefinitionen im Namespace System.Collections.Generic

Interface	Beschreibung
`IEnumerable`	Gibt den Enumerator zum Durchlaufen der Collection mittels der Methode `GetEnumerator()` bekannt.
`IEnumerator`	Stellt folgende Methoden und Eigenschaften für den Enumerator zur Verfügung (wird benötigt für die `For Each`-Schleife): ■ Current ■ MoveNext() ■ Reset()
`IEqualityComparer`	Stellt folgende beiden Methoden zur Verfügung, um Objekte auf Gleichheit zu überprüfen: ■ Equals() ■ GetHashCode()
`IList`	Stellt folgende Methoden und Eigenschaften zur Verfügung, um auf Objekte über Indizes zuzugreifen: ■ IndexOf() ■ Insert() ■ Item (Eigenschaft) ■ RemoveAt() Implementiert außerdem die Interfaces `IEnumerable` und `ICollection`.

In der Tabelle 6.3 will ich noch kurz auflisten, welche Klassen aus Tabelle 5.2 welche Interfaces aus Tabelle 5.3 implementieren.

Tabelle 5.4 Übersicht der in den Generic-Klassen implementierten Interfaces

Klasse	Implementierte Interfaces
`Dictionary`	`IDictionary`, `ICollection`, `IEnumerable`
`LinkedList`	`ICollection`, `IEnumerable`
`List`	`IList`, `ICollection`, `IEnumerable`
`Queue`	`IEnumerable`
`SortedDictionary`	`IDictionary`, `ICollection`, `IEnumerable`
`SortedList`	`IDictionary`, `ICollection`, `IEnumerable`
`Stack`	`IEnumerable`

Somit haben Sie hoffentlich eine Vorstellung, welchen Funktionsumfang die bestimmten Generic-Auflistungsklassen besitzen.

Nun will ich Ihnen noch in einer kleinen Applikation die Syntax für Generics am Beispiel einer `SortedList` zeigen.

Dazu habe ich eine Klasse `Person` geschrieben, die nur drei Eigenschaften `Vorname`, `Famname` und `Geburtsdatum` besitzt. Der Konstruktor dieser Klasse erwartet zwei Parameter vom Typ `String` für den Vor- und Familiennamen.

In meiner Applikation definiere und instanziiere ich ein Objekt `Personen` vom Typ `SortedList`.

```
Private Personen As New
    System.Collection.Generic.SortedList(Of String, Person)
```

Wie Sie sehen, wird mittels eines `Of`-Operators dieser Klasse mitgeteilt, von welchem Datentyp der Schlüssel und von welchem Datentyp die in dieser Auflistung enthaltenen Objekte sind.

Ich kann somit nur noch Objekte vom Typ `Person` oder von `Person` abgeleitete Typen, die als Schlüssel einen `String` besitzen, zu dieser Collection hinzufügen. Mehr muss ich nicht tun.

Danach füge ich der Auflistung zwei Personenobjekte hinzu. Würde ich hier einen Schlüssel angeben, der nicht vom Datentyp `String` ist, oder versuchen, ein anderes Objekt hinzuzufügen, würde ein Compiler-Fehler erzeugt werden und kein Laufzeitfehler.

Wenn ich nun über die `Items`-Eigenschaft auf ein bestimmtes Personenobjekt zugreifen will, sehen Sie in Abbildung 5.16, dass ich keine Typumwandlung durchführen muss und außerdem eine vollständige IntelliSense-Unterstützung habe.

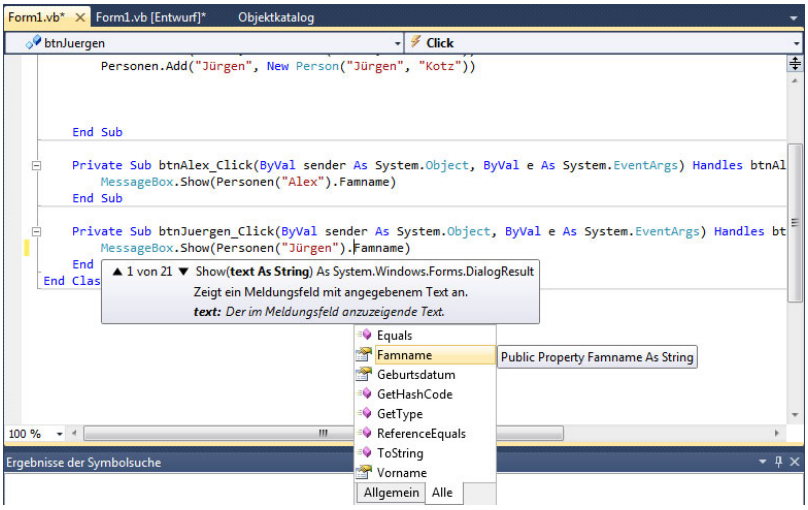


Abbildung 5.16
Intellisense-Unterstützung bei Generics

Den kompletten Programmcode zu dem Beispiel sehen Sie in Listing 5.36.

```
Public Class Form1
    Private Personen As New
     System.Collections.Generic.SortedList(Of String, Person)

    Private Sub Form1_Load(ByVal sender As Object,
        ByVal e As System.EventArgs) Handles MyBase.Load

        Personen.Add("Alex", New Person("Alex", "Bierhaus"))
        Personen.Add("Jürgen", New Person("Jürgen", "Kotz"))
    End Sub
```

Listing 5.36
Beispielcode für Generics

Listing 5.36 (Forts.)
Beispielcode für Generics

```
    Private Sub btnAlex_Click(ByVal sender As Object,
        ByVal e As System.EventArgs) Handles btnAlex.Click

        MessageBox.Show(Personen("Alex").Famname)
    End Sub

    Private Sub btnJuergen_Click(ByVal sender As Object,
        ByVal e As System.EventArgs) Handles btnJuergen.Click

      MessageBox.Show(Personen("Jürgen").Famname)
    End Sub
End Class

Public Class Person
    Private mVorname As String
    Private mFamname As String
    Private mGeburtsdatum As Date

    Public Sub New(ByVal vorName As String, _
        ByVal famName As String)
        Me.Vorname = vorName
        Me.Famname = famName
    End Sub

    Public Property Vorname() As String
        Get
            Return mVorname
        End Get
        Set(ByVal value As String)
            mVorname = value
        End Set
    End Property

    Public Property Famname() As String
        Get
            Return mFamname
        End Get
        Set(ByVal value As String)
            mFamname = value
        End Set
    End Property

    Public Property Geburtsdatum() As Date
        Get
            Return mGeburtsdatum
        End Get
        Set(ByVal value As Date)
            mGeburtsdatum = value
        End Set
    End Property
End Class
```

Dieses Beispiel soll zeigen, wie einfach grundsätzlich Generics anzuwenden sind und warum Sie diesen Namespace auch nutzen sollten, wenn Sie mit Auflistungsklassen arbeiten.

Nicht zu unterschätzen ist wirklich der Performancegewinn gegenüber den herkömmlichen Collection-Klassen. Da der Compiler den Code bezüglich der benutzten Typen optimiert, werden keine laufzeitschädlichen Typumwandlungen mehr durchgeführt.

Info

Generics verwenden auch kein **Boxing** und **Unboxing**. Unter Boxing versteht man das Umwandeln eines Wertetyps in einen Referenztyp, Unboxing ist die Rückumwandlung zurück in einen Wertetyp. Dieser Vorgang wird bei den herkömmlichen Auflistungsklassen durchgeführt, wenn einfache Typen gespeichert werden müssen, da diese zu einem `Object` und danach wieder zurück konvertiert werden müssen.

Doch die ganze Sache geht noch einen Schritt weiter. Denn Sie können nicht nur die Generic-Klassen nutzen, Sie können auch selbst eigene Typen und Methoden erstellen.

Generische Methoden sind zum Beispiel dann sinnvoll, wenn Sie mittels einer Methode, unabhängig vom Datentyp der übergebenen Parameter, einen bestimmten Algorithmus anwenden wollen.

Ein klassisches Beispiel hierfür ist ein **BubbleSort**-Algorithmus.

Bei diesem Algorithmus werden jeweils zwei Werte miteinander verglichen und bei Bedarf getauscht. Und genau die Routine, in der die Werte vertauscht werden, wollen wir als generische Methode definieren. Dies hat den Vorteil der Wiederverwendbarkeit des geschriebenen Codes für unterschiedliche Datentypen.

Innerhalb einer Konsolenanwendung generiere ich per Zufallsgenerator ein Array mit zehn Zufallszahlen vom Typ `Integer`. Mittels des BubbleSort-Algorithmus will ich die Werte sortieren und verwende dazu eine Methode `Wechsle()`, die zwei Werte bei Bedarf miteinander tauscht. Die Methode `Wechsle()` ist dabei als generische Methode definiert. Das Beispiel finden Sie in Listing 5.37.

Listing 5.37
Beispiel für eine generische Methode

```
Sub Main()
    Dim zahlen(9) As Integer
    Dim r As New Random
    'Zahlengenerierung
    For i As Integer = 0 To 9
        zahlen(i) = r.Next(0, 100)
    Next
    'Sortierung
    For i As Integer = 0 To 9
        For j As Integer = 9 To 1 Step -1
            If zahlen(j) < zahlen(j - 1) Then
                Wechsle(Of Integer)(zahlen(j), zahlen(j - 1))
            End If
        Next
    Next
    'Ausgabe
    For i As Integer = 0 To 9
        Console.WriteLine(zahlen(i))
    Next
```

Listing 5.37 (Forts.) Beispiel für eine generische Methode

```
    Console.ReadLine()
End Sub

Private Sub Wechsle(Of ItemType) (ByRef item1 As ItemType,
     ByRef item2 As ItemType)

    Dim temp As ItemType
    temp = item1
    item1 = item2
    item2 = temp
End Sub
```

An den `Of`-Operator wird ein beliebiger Typ übergeben. Die Bezeichnung `ItemType` wird beim Aufruf an jeder Stelle innerhalb des Programmcodes durch den tatsächlich übergebenen Typ ersetzt. Der Datentyp muss dabei vom Aufrufer zusätzlich zu den beiden anderen Parametern übergeben werden, wie Sie in der `Main()`-Routine sehen können.

```
Wechsle(Of Integer)(zahlen(j), zahlen(j - 1))
```

Der JIT-Compiler generiert bei diesem Aufruf folgenden Programmcode:

```
Private Sub Wechsle(ByRef item1 As Integer,
  ByRef item2 As Integer)
    Dim temp As Integer
    temp = item1
    item1 = item2
    item2 = temp
End Sub
```

Wäre der Aufruf mit `Of String` erfolgt, so hätte der JIT-Compiler an den entsprechenden Stellen `String`-Variablen verwendet.

Genauso gut können Sie auf dieselbe Art und Weise auch eigene generische Typen definieren:

```
Public Class MyGenericClass(Of ItemType)
```

Wobei an dieser Stelle `ItemType` an jeder beliebigen Stelle innerhalb dieser Klasse als Datentyp verwendet werden kann, der dann vom Compiler durch den an den `Of`-Operator übergebenen Typ ersetzt wird.

Generics waren aus meiner Sicht das Highlight der neuen Features in .NET 2.0, denn sie bieten wesentliche Verbesserungen bezüglich Laufzeitverhalten, strikter Typenüberprüfung und IntelliSense-Unterstützung gegenüber den bislang etablierten Collection-Objekten.

5.8 Das My-Object

Das `My`-Object wurde mit Visual Basic 8 eingeführt. Mit `My` hat der Visual Basic-Entwickler einen einfachen und schnellen Zugriff auf wichtige Klassen, die ansonsten in den Tiefen der Framework-Klassenbibliothek verstreut sind.

Info

Im Gegensatz zu Generics steht `My` nur in Visual Basic zur Verfügung, für C# oder andere .NET-Sprachen wurde es nicht implementiert.

`My` bietet Zugriff auf folgende Objekte:

- Application

  `My.Application` bietet Zugriff auf Daten, die der aktuellen Applikation zugeordnet sind (wie zum Beispiel Kulturinformationen, Befehlszeilenargumente, alle geöffneten Formulare).

- Computer

  `My.Computer` bietet Zugriff auf Computerkomponenten (wie zum Beispiel Zwischenablage, Uhr, Tastatur, Dateisystem).

- Forms

  `My.Forms` bietet Zugriff auf ein instanziiertes Formular innerhalb einer Windows-Applikation. `My.Forms` ist auch nur in Windows Forms-Anwendungen verfügbar.

- Resources

  `My.Resources` bietet schreibgeschützten Zugriff auf Ressourcen Ihrer Applikation (wie zum Beispiel Bilder und Zeichenfolgen für Lokalisierung).

- Settings

  `My.Settings` bietet lesenden und schreibenden Zugriff auf Anwendungseinstellungen.

- User

  `My.User` bietet Zugriff auf Informationen des aktuellen Benutzers (wie zum Beispiel Name, Windows-Gruppen).

- WebServices

  `My.WebServices` bietet Zugriff auf jede Webservice-Instanz, auf die aus Ihrem Projekt referenziert wird.

Im Folgenden will ich zu einigen `My`-Objekten ein kleines Beispiel zeigen, um einen Überblick über die Mächtigkeit dieses Konstrukts zu bieten. Dabei werden in einer `ListView` verschiedene Einträge über das entsprechende Objekt in die Liste eingetragen.

Beginnen wir mit `My.Application` und Listing 5.38

Listing 5.38
Beispiel für My.Application

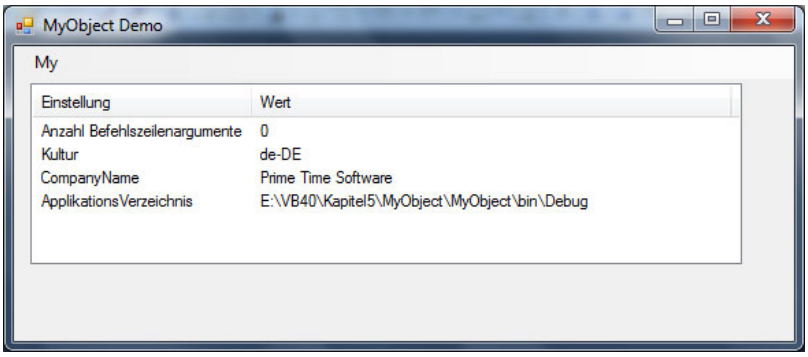
```
Dim li As New ListViewItem
li.SubItems.Add(My.Application.CommandLineArgs.Count.
   ToString())
li.Text = "Anzahl Befehlszeilenargumente"
ListView1.Items.Add(li)

li = New ListViewItem()
li.Text = "Kultur"
li.SubItems.Add(My.Application.Culture.ToString)
ListView1.Items.Add(li)

li = New ListViewItem
li.Text = "CompanyName"
li.SubItems.Add(My.Application.Info.CompanyName)
ListView1.Items.Add(li)
```

Listing 5.38 (Forts.)
Beispiel für My.Application

```
li = New ListViewItem
li.Text = "ApplikationsVerzeichnis"
li.SubItems.Add(My.Application.Info.DirectoryPath)
ListView1.Items.Add(li)
```

Das Ergebnis sehen Sie in Abbildung 5.17.

Abbildung 5.17
My.Application



Bei den folgenden Codebeispielen will ich nur noch den Code darstellen, der den Aufruf an das `My`-Objekt durchführt.

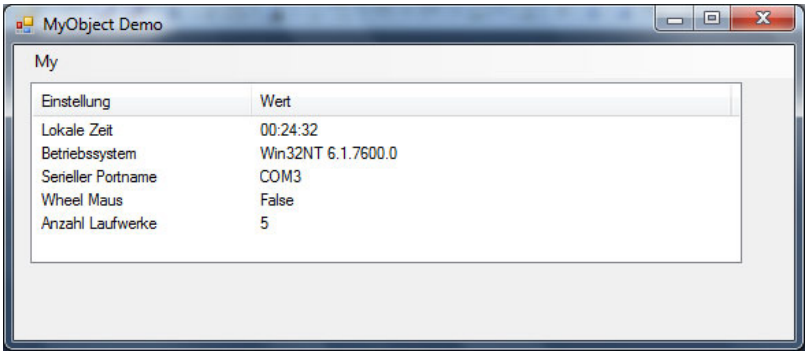
Listing 5.39 zeigt ein Beispiel für `My.Computer`.

Listing 5.39
Beispiel für My.Computer

```
li.SubItems.Add(My.Computer.Clock.LocalTime.
    ToLongTimeString)
li.SubItems.Add(My.Computer.Info.OSPlatform &
    " " & My.Computer.Info.OSVersion)
li.SubItems.Add(
   My.Computer.Ports.SerialPortNames(0).ToString)
li.SubItems.Add(
   My.Computer.Mouse.WheelExists.ToString)
li.SubItems.Add(
   My.Computer.FileSystem.Drives.Count.ToString)
```

Abbildung 5.18 zeigt die Ausgabe für `My.Computer`.

Abbildung 5.18
My.Computer



Nur in Windows Forms-Applikationen verfügbar ist `My.Forms`.

```
li.SubItems.Add(My.Forms.Formulartest.Text)
li.SubItems.Add(
   My.Forms.Formulartest.IsMdiContainer.ToString)
li.SubItems.Add(
   My.Forms.Formulartest.BackColor.G.ToString)
li.SubItems.Add(
   My.Forms.Formulartest.Controls.Count.ToString)
```

Listing 5.40
Beispiel für My.Forms

Hierfür habe ich ein zweites Formular `Formulartest` zur Applikation hinzugefügt. Abbildung 5.19 zeigt die entsprechende Ausgabe.

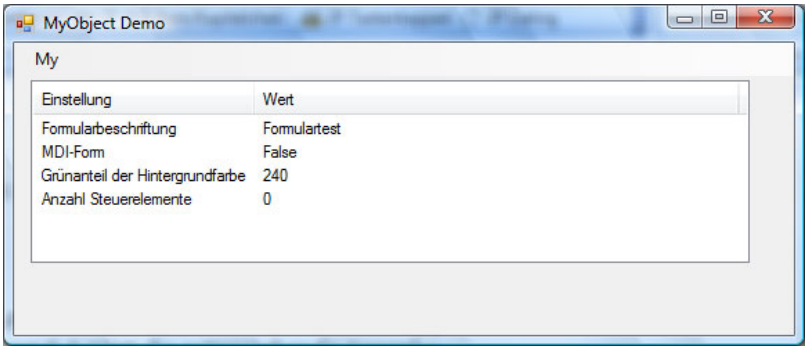


Abbildung 5.19
My.Forms

Und zu guter Letzt noch ein Beispiel für `My.User`.

```
li.SubItems.Add(My.User.Name)
li.SubItems.Add(My.User.IsInRole
    (ApplicationServices.BuiltInRole.Administrator.ToString))
li.SubItems.Add(My.User.CurrentPrincipal.
    Identity.AuthenticationType.ToString)
li.SubItems.Add(My.User.CurrentPrincipal.
    Identity.IsAuthenticated)
```

Listing 5.41
Beispiel für My.User

Was zur Bildschirmausgabe in Abbildung 5.20 führt.

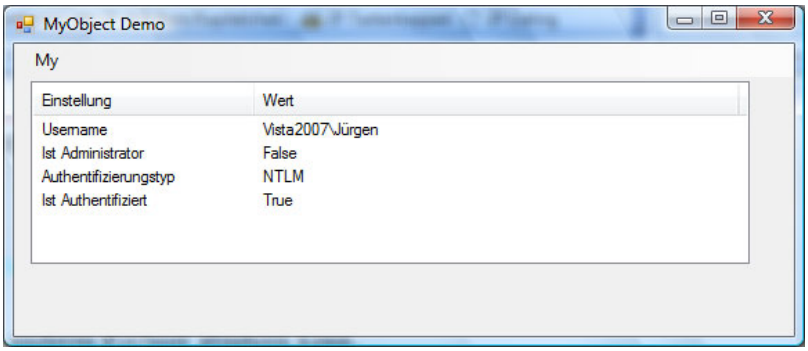


Abbildung 5.20
My.User

Zu erwähnen ist, dass es über das My-Objekt nicht nur möglich ist, Eigenschaften aus zentralen Stellen auszulesen oder zu setzen, sondern auch Funktionalität auszuführen.

Die Anweisung

```
My.Computer.FileSystem.CreateDirectory("C:\test")
```

würde zum Beispiel ein neues Verzeichnis *test* anlegen, wobei `My.Computer.FileSystem` einen vollständigen Zugriff (falls die Sicherheitseinstellungen es zulassen) auf das Dateisystem des Rechners gewährt.

Während man mit

```
My.Computer.Network.Ping(InputBox(InputBox("IP-Adresse"))
```

einen **Ping** auf einen anderen Rechner absetzen kann.

Die Möglichkeiten sind sehr vielfältig und erleichtern an vielen Stellen die Arbeit wirklich enorm.

5.9 Operator Overloading

Mittels des Überladens von Operatoren können Sie in einer Klasse das Verhalten von zukünftigen Objekten definieren, wenn diese im Code mit den entsprechenden Operatoren notiert werden. Dass durch ein + zum Beispiel zwei Strings konkateniert werden, ist sicherlich jedem einleuchtend, aber was passiert, wenn Sie den +-Operator für Objekte selbst definierter Klassen einsetzen? Mittels des Überladens von Operatoren können Sie für Ihre eigenen Klassen bestimmte Funktionalität hinterlegen.

Sie können unter anderem die folgenden Operatoren überladen: +, –, <, <=, >, >=, =, <>, *, /.

Nehmen wir an, wir haben eine Artikelklasse und wollen ein Array von Artikeln nach Preisen sortieren. Dazu wollen wir denselben Bubble-Sort-Algorithmus wie in Listing 5.37 verwenden (inklusive unserer generischen `Wechsle()`-Methode).

Um das zu realisieren, müssen wir den >- und <-Operator für die Klasse `Artikel` überschreiben, wobei das Definieren von überladenen Operatoren eigentlich ganz einfach ist.

Sie machen das, als würden Sie eine herkömmliche Methode anlegen, anstatt `Sub` oder `Function` schreiben Sie jedoch `Operator` und statt den Methodennamen geben Sie den Operator an. Der Operator muss zwingend als statische Methode mit dem Schlüsselwort `Shared` gekennzeichnet sein und einer der übergebenen Parameter muss vom Typ der entsprechenden Klasse sein. Wenn es mehrere Parameter gibt (was bei Vergleichsoperatoren wie <, >, =, >=, <= ja durchaus Sinn macht), muss zumindest einer von beiden vom entsprechenden Typ sein.

Innerhalb des überladenen Operators schreiben Sie einfach die Logik, die Sie an dieser Stelle einsetzen wollen.

Achtung

Achten Sie bitte darauf, dass Sie innerhalb der Logik die Parameter auch auf `Nothing` überprüfen, da eventuell eine Objektvariable noch nicht zugewiesen ist und Sie an dieser Stelle keinen Laufzeitfehler riskieren wollen.

Bestimmte Operatoren lassen sich nicht allein überladen. Wenn Sie zum Beispiel den <-Operator überladen, müssen Sie auch den >-Operator überladen, was durchaus seine Konsequenz hat. Machen Sie das nicht, resultiert die ganze Sache in einem Compiler-Fehler.

Im Listing 5.42 sehen Sie die Klassendefinition für eine Artikelklasse inklusive Überladung für die Operatoren <, >, =, <>.

Listing 5.42
Beispiel für Operator Overloading

```
Public Class Artikel
    Private mBezeichnung As String
    Private mPreis As Double
    Public Property Bezeichnung() As String
        Get
            Return mBezeichnung
        End Get
        Set(ByVal value As String)
            mBezeichnung = value
        End Set
    End Property
    Public Property Preis() As Double
        Get
            Return mPreis
        End Get
        Set(ByVal value As Double)
            mPreis = value
        End Set
    End Property

    Public Shared Operator >(ByVal Artikel1 As Artikel,
        ByVal Artikel2 As Artikel) As Boolean

        If Artikel1 Is Nothing AndAlso Artikel2 Is Nothing Then
            Return False
        End If
        If Artikel1 Is Nothing Then
            Return False
        End If
        If Artikel2 Is Nothing Then
            Return True
        End If
        Return Artikel1.Preis > Artikel2.Preis
    End Operator

    Public Shared Operator <(ByVal Artikel1 As Artikel,
        ByVal Artikel2 As Artikel) As Boolean

        If Artikel1 Is Nothing AndAlso Artikel2 Is Nothing Then
            Return False
```

Listing 5.42 (Forts.) Beispiel für Operator Overloading

```
            End If
            If Artikel1 Is Nothing Then
                Return True
            End If
            If Artikel2 Is Nothing Then
                Return False
            End If
            Return Artikel1.Preis < Artikel2.Preis
        End Operator

        Public Shared Operator =(ByVal Artikel1 As Artikel,
            ByVal Artikel2 As Artikel) As Boolean

            If Artikel1 Is Nothing AndAlso Artikel2 Is Nothing Then
                Return True
            End If
            If Artikel1 Is Nothing Then
                Return False
            End If
            If Artikel2 Is Nothing Then
                Return False
            End If
            Return Artikel1.Preis = Artikel2.Preis
        End Operator

        Public Shared Operator <>(ByVal Artikel1 As Artikel,
            ByVal Artikel2 As Artikel) As Boolean

            If Artikel1 Is Nothing AndAlso Artikel2 Is Nothing Then
                Return False
            End If
            If Artikel1 Is Nothing Then
                Return True
            End If
            If Artikel2 Is Nothing Then
                Return True
            End If
            Return Artikel1.Preis <> Artikel2.Preis
        End Operator
    End Class
```

In der `Sub Main()` der Konsolenanwendung wollen wir jetzt diese Operatorenüberladung nutzen und die Artikel nach Preis sortieren. Den Programmcode sehen Sie in Listing 5.43.

Listing 5.43 Operator Overloading in der Praxis

```
Dim Produkte(9) As Artikel
Dim r As New Random
'Array befüllen
For i As Integer = 0 To 9
    Dim a As New Artikel
    a.Bezeichnung = "Artikel " & i.ToString
    a.Preis = r.NextDouble() * 100
    Produkte(i) = a
Next
'Sortierung
For i As Integer = 0 To 9
    For j As Integer = 9 To 1 Step -1
        If Produkte(j) < Produkte(j - 1) Then
```

```
            Wechsle(Of Artikel)(Produkte(j), Produkte(j - 1))
        End If
    Next
Next
'Ausgabe
For i As Integer = 0 To 9
    Console.WriteLine(Produkte(i).Bezeichnung &
    "  " & Produkte(i).Preis.ToString("##0.00"))
Next
Console.ReadLine()
```

Listing 5.43 (Forts.) Operator Overloading in der Praxis

Wir können jetzt die Produkte mit den von uns überladenen Operatoren bequem vergleichen. Ich finde es auch schön, wie unsere generische `Wechsle()`-Methode ebenso für Artikel funktioniert.

Sicherlich hätten wir dieses Problem auch ohne das Überladen von Operatoren relativ leicht lösen können, doch ich wollte das Beispiel bewusst einfach halten. Und je aufwändiger die dahinter liegende Logik ist, desto mehr macht es Sinn, dieses Feature auch einzusetzen. Beachten Sie bitte nur, dass das Lesen des Programmcodes auch durch den Einsatz dieser Technologie noch intuitiv bleiben soll, denn Code wird bei weitem öfter gelesen als geschrieben. Also übertreiben Sie es nicht mit dem Operator Overloading.

5.9.1 Überladen von CType

Eine spezielle Art der Operatorenüberladung gibt es für den `CType`-Operator.

Sie können definieren, wie Ihr Objekt in jeden beliebigen einfachen Datentyp gewandelt wird, indem Sie den `CType`-Operator erweitern.

Ich habe für unsere Artikelklasse eine Konvertierung auf `String` und auf `Double` erweitert, so dass, wenn ich einen `Artikel` in einen `String` umwandle, die Artikelbezeichnung zurückgegeben wird und bei einer Umwandlung nach `Double` der entsprechende Preis des Artikels.

Listing 5.44 zeigt die Erweiterung der Artikelklasse.

```
Public Shared Widening Operator CType
       (ByVal art As Artikel) As String

    If art Is Nothing Then
        Return ""
    Else
        Return art.Bezeichnung
    End If
End Operator

Public Shared Widening Operator CType
    (ByVal art As Artikel) As Double

    If art Is Nothing Then
        Return 0.0
    Else
        Return art.Preis
    End If
End Operator
```

Listing 5.44 Erweiterung der CType-Überladung

Mit dem Schlüsselwort `Widening` wird die Überladung für den `CType`-Operator erweitert.

Sie können jetzt eine Objektinstanz von Artikeln mit folgender Anweisung umwandeln:

```
Dim x As String = CType(artikel, String)
```

oder

```
Dim y As Double = CType(artikel, Double)
```

5.9.2 IsTrue- und IsFalse-Operatoren

Interessant ist an dieser Stelle auch noch der Hinweis auf die Überladungsmöglichkeit für zwei Operatoren, die Sie im Code direkt gar nicht benutzen dürfen und die Ihnen deswegen wohl auch nicht bekannt sind.

`IsTrue` und `IsFalse` sind zwei Operatoren, die nur für Überladung zur Verfügung stehen.

Mittels dieser Überladungen ist es für den Compiler möglich, einen booleschen Wert zu bestimmen. Dies geschieht in dem Fall, dass ein Objekt an einer Stelle benutzt wird, an der ein boolescher Wert erwartet wird, wie zum Beispiel in folgender Abfrage `If art Then`.

Bislang würde der Compiler einen Fehler melden und zwar, dass der Typ nicht zu `Boolean` konvertiert werden kann. Wenn Sie jedoch die `IsTrue`- und `IsFalse`-Operatoren überladen, wird der Compiler genau dies können.

Listing 5.45 zeigt, wie Sie diese Überladung im Code implementieren können.

Listing 5.45 Überladung von IsTrue und IsFalse

```
Public Shared Operator IsTrue (ByVal art As Artikel) As Boolean

    If art Is Nothing Then
        Return False
    Else
        Return True
    End If
End Operator
Public Shared Operator IsFalse
       (ByVal art As Artikel) As Boolean

    If art Is Nothing Then
        Return True
    Else
        Return False
    End If
End Operator
```

Das bedeutet, wenn jetzt die Objektvariable auf einen booleschen Wert abgefragt wird, ist der Rückgabewert in Abhängigkeit, ob die Variable bereits einer Instanz zugewiesen ist, `true` oder `false`. Dank dieser Überladung brauchen Sie jetzt eigentlich nicht mehr auf `Is Nothing` abzuprüfen, das erledigt diese Überladung für Sie. Und wenn wir jetzt schon dabei sind, überladen wir auch gleich noch den `Not`-Operator, wie Sie in Listing 5.46 sehen.

Listing 5.46
Überladung des Not-Operators

```
Public Shared Operator Not(ByVal art As Artikel) As Boolean
    If art Then
        Return False
    Else
        Return True
    End If
End Operator
```

So, das wär´s gewesen zu Operator Overloading und jetzt schauen wir uns im nächsten Kapitel die neuen Sprachmerkmale in Visual Basic 10 an.

6

Neue Features in Visual Basic 10

In diesem Kapitel möchte ich die wichtigsten Neuigkeiten der Version Visual Basic 10 vorstellen.

6.1 Sprachneuerungen in Visual Basic 10

Mit der Sprachversion 10 wurden einige kleine Spracherweiterungen eingeführt, die ich in diesem Abschnitt kurz vorstellen will.

6.1.1 Dynamische Spracherweiterungen

In letzter Zeit gab es immer mehr Sprachen, wie zum Beispiel JavaScript, IronRuby oder IronPython, auf dem Markt, die mit dynamischer Programmierung sehr viel Erfolg hatten. Die Möglichkeiten, die dynamische Sprachen besitzen, wurden jetzt auch in VB 10 eingeführt.

Bevor wir uns jedoch mit der Implementierung in VB befassen, sollten wir uns kurz Gedanken machen, wo die Vorteile von dynamischen Sprachen liegen.

Dynamische Sprachen können mit zur Kompilierzeit unbekannten Datentypen wesentlich intuitiver und leichter umgehen, als dies mit statischen Sprachen möglich wäre.

Im .NET Framework gibt es mit Reflections auch eine Zauberbox, die mit unbekannten Typen zur Kompilierzeit gut umgehen kann, doch ob der resultierende Code immer intuitiv und gut les- und wartbar ist, sei einmal dahingestellt.

Sie werden erkennen, dass sie mit den dynamischen Möglichkeiten auf viele Typcasts verzichten können, was wie bei den gerade beschriebenen optionalen Parametern, gerade bei der Office-Automation ein wahrer Segen ist.

Es sollte jedoch auch nicht verschwiegen werden, dass diese Dynamik auch ihren Preis hat. Bei statischen Sprachen muss der verwendete Datentyp bereits zur Kompilierzeit bekannt sein und somit können Tippfehler oder sonstige syntaktische Fehler bereits zur Kompilierzeit erkannt werden. Die Typenüberprüfung bei dynamischen Sprachen erfolgt erst zur Laufzeit, was natürlich in Bezug auf Fehleranfälligkeit ein Nachteil ist. Somit kann ein statischer Compiler natürlich auch Optimierungen am Code ausführen, was in der Regel zu einer bei weitem besseren Laufzeit gegenüber dynamischen Sprachen führt.

Visual Basic 10 verwendet dazu den Datentyp `Object`, mit dem die dynamischen Möglichkeiten auch in VB Einzug halten.

```
Dim test As Object = "Dynamisch"
```

Bitte beginnen Sie jetzt nicht, dieses Konzept mit impliziter Typisierung zu vergleichen. Es handelt sich hierbei wirklich um zwei verschiedene Paar Stiefel. Die implizite Typisierung, die ohne Angabe eines Datentyps einhergeht, stellt den entsprechenden Typ bereits zur Kompilierzeit fest und der Compiler erzeugt den fehlenden Code. Die Vorteile sind IntelliSense, ein besseres Laufzeitverhalten und Typchecks bereits zur Kompilierungszeit. Dynamische Sprachen werten dagegen den Ausdruck `Object` erst zur Laufzeit aus.

Schauen wir uns einfach mal ein Beispiel an, um Reflections und `Dynamische Programmierung` miteinander zu vergleichen. Dabei erzeugen wir einmal mittels Reflections und einmal mit dem dynamischen Sprachkonstrukt ein Objekt vom Typ `Random` und dann rufen wir die Methode `Next()` dieses Objekts auf.

Listing 6.1 Beispiel für dynamische Programmierung im Vergleich zu Reflections

```
Sub Form_Load(ByVal sender As Object, ByVal e As EventArgs)
   'Reflections
    Dim zufallsGenerator As Object =
      Activator.CreateInstance(Type.GetType("System.Random"))
    Dim objType As Type = zufallsGenerator.GetType()
    Dim method = objType.GetMethod("Next",
        System.Type.EmptyTypes)
    MessageBox.Show(method.Invoke(zufallsGenerator,
             Nothing).ToString())
    'Dynamic
    Dim dynamicGenerator As Object =
        Activator.CreateInstance(Type.GetType("System.Random"))
    MessageBox.Show(dynamicGenerator.Next().ToString())
  End Sub
```

Wir denken, der Unterschied in der Lesbarkeit ist offensichtlich.

Die Verwendung dieses dynamischen Sprachkonstrukts macht in den folgenden Szenarien Sinn:

- Beim Arbeiten mit COM-Objekten, um viele Typumwandlungen zu vermeiden und somit lesbareren Code zu schreiben
- Bei der Interaktion mit dynamischen Sprachen
- Bei der Arbeit mit Objekten, bei denen sich die Strukturen sehr häufig ändern können, wie zum Beispiel XML-Dokumente

6.1.2 Auflistungsinitialisierer

Mittels Auflistungsinitialisierer wird eine Kurzsyntax zur Verfügung gestellt, bei der eine Auflistung erstellt und gleichzeitig mit Werten initialisiert werden kann.

Der folgende Codeausschnitt zeigt dabei die Initialisierung einer Liste von Strings.

```
Dim namen As New List(Of String) From
   {"Christian",  "Tobias", "Karsten", "Jürgen"}
```

Das Ganze funktioniert natürlich auch mit komplexeren Typen wie im Folgenden dargestellt ist.

Listing 6.2
Auflistungsinitialisierer mit komplexen Typen

```
Dim customers As New List(Of Person) From
 {New Person() With {.Name = "Wenz", .Vorname = "Christian"},
 New Person() With {.Name = "Hauser", .Vorname = "Tobias"},
 New Person() With {.Name = "Samaschke", .Vorname = "Karsten"},
 New Person() With {.Name = "Kotz", .Vorname = "Jürgen"}
 }
```

6.1.3 Implizite Zeilenfortsetzung

Bislang musste jede Anweisung in Visual Basic als eine einzige Zeile geschrieben werden. Ein *;* wie in C# als Endezeichen für die Anwendung gibt es in Visual Basic nicht. Damit bei langen Anweisungen der Quellcode noch lesbar bleibt, konnte man mit dem _-Zeichen einen Zeilenumbruch definieren und die Anweisung in der nächsten Zeile fortsetzen.

Dieses Zeilenfortsetzungszeichen ist in Visual Basic 10 jetzt nicht mehr zwingend nötig, da der Compiler bei folgenden Zeichen selbst erkennt, dass ein Zeilenumbruch erfolgt:

- Nach einem Komma oder Punkt
- Nach einem > (Ende eines Attributs)
- Öffnende Klammern (oder {
- Vor einer schließenden Klammer), },]
- Vor oder nach jedem LINQ-Schlüsselwort
- Nach jedem Operator
- Nach dem With-Ausdruck bei der Objektinitialisierung

6.1.4 Array-Literale

Visual Basic 10 erleichtert die Definition von Arrays durch eine implizite Typisierung basierend auf sogenannten Array-Literalen.

In der Vergangenheit konnten Sie ein Integer-Array wie folgt definieren und initialisieren:

```
Dim zahlen As Integer() = New Integer() { 1, 2, 3, 4}
```

Eigentlich kann der Compiler aufgrund der Werte erkennen, dass es sich um ein Array von Ganzzahlen handelt. Deswegen reicht in Visual Basic 10 bereits folgende Definition:

```
Dim zahlen = { 1, 2, 3, 4}
```

6.1.5 Weitere Sprachneuerungen

Des Weiteren gibt es noch folgende Sprachneuerungen:

- Optionale Parameter benötigen keinen Wert mehr, können also mit Nothing vorinitialisiert werden
- Einbetten von Interoptypen für Office-Automatisierung. Dadurch müssen auf den Zielrechnern nicht mehr zwingend die Primary Interop Assemblies (PIAs) zur Verfügung stehen.
- Verbesserungen und Vereinfachungen beim Entwickeln von Multithreading-Anwendungen (mehr dazu in Kapitel 12)

6.2 MEF – Managed Extensibility Framework

MEF wurde mit dem .NET Framework 4 eingeführt und bietet ein sehr umfangreiches Framework, um anpassungsfähige Applikationen zu schreiben.

Somit können Kunden bestehende Anwendungen erweitern und ihre eigene Geschäftslogik implementieren, ohne dass dazu Änderungen am ursprünglichen Programm durchgeführt werden müssen.

Solche Erweiterungen werden ***Extensions*** genannt und MEF bietet dazu unterschiedliche Wege, diese zu laden.

Betrachten wir aber nun einmal, wie wir ein kleines MEF-Beispiel zur Veranschaulichung erstellen.

6.2.1 MEF – erste Schritte

Erstellen Sie dazu eine neue Konsolenanwendung und fügen Sie einen Verweis auf die Bibliothek ***System.ComponentModel.Composition*** hinzu. In dieser Bibliothek ist die Grundfunktionalität von MEF enthalten.

Fügen Sie anschließend der Projektmappe eine Klassenbibliothek hinzu. Diese Klassenbibliothek besteht nur aus einer einzigen Interfacedefinition. Auf diese Art und Weise können Sie Ihrem Kunden das Interface, separiert vom Rest der Applikation, bereitstellen, damit dieser basierend auf diesem Interface eigene Geschäftslogik implementieren kann.

Listing 6.3 Definition des Interface Itest

```
Public Interface ITest
    Property Message As String
End Interface
```

Listing 6.3 zeigt die Definition des Interface, das nur aus einer einzigen Property ***Message*** besteht.

Im nächsten Schritt fügen wir unserer Konsolenapplikation eine Klasse hinzu, die das entsprechende Interface implementiert und deren Implementierung vom Kunden erweitert werden kann.

Vergessen Sie bitte nicht, der Konsolenanwendung einen Verweis auf die Interfacebibliothek hinzuzufügen.

Die neue Klasse sehen Sie in Listing 6.4.

Listing 6.4
Implementierung des Interface in der Klasse Test

```
Imports System.ComponentModel.Composition
Imports System.ComponentModel.Composition.Hosting
Public Class Test
    Implements IMEF.ITest
    <Import()>
    Public Property Message As String _
           Implements IMEF.ITest.Message
End Class
```

Die Eigenschaft *Message* wurde zusätzlich mit dem Attribut *Import* versehen. Dieses Attribut kommt eben aus der Bibliothek *System.ComponentModel.Composition*, die wir zu Beginn als Projektverweis der Konsolenapplikation hinzugefügt haben.

Durch dieses Attribut definieren Sie diese Funktion als erweiterbar.

Ich habe an dieser Stelle ganz bewusst eine Property verwendet und keine Funktion, da das *Import*-Attribut für Funktionen nicht erlaubt ist.

Als Nächstes wollen wir die Property *Message*, die ja noch ohne Implementierung ist, in einer weiteren Klasse erweitern. Diese Aufgabe wird in der Regel Ihr Kunde übernehmen, um kundenspezifische Erweiterungen zu implementieren. Der Einfachheit halber werden wir die Klasse in das Konsolenprojekt integrieren. Später und in der Realität wird es sich dabei sehr wohl um eine eigene Bibliothek handeln.

Listing 6.5 zeigt die Implementierung der Klasse *TestExtension*.

Listing 6.5
Implementierung der Erweiterungsklasse

```
Imports System.ComponentModel.Composition
Public Class TestExtension

    <Export()>
    Public Property Message As String
        Get
            Return "Hello Visual Basic 10"
        End Get
        Set(ByVal value As String)
        End Set
    End Property
End Class
```

Diese Property wurde mit dem *Export*-Attribut versehen. Das *Export*-Attribut ist das Gegenstück zum *Import*-Attribut.

So weit, so gut. Jetzt wird es aber erst interessant. Da das Managed Extensibility Framework mehrere Möglichkeiten zur Verfügung stellt, wie die Erweiterung geladen werden kann, muss die gewünschte Möglichkeit dem Programm auch mitgeteilt werden.

Dazu fügen Sie bitte der *Test*-Klasse die in Listing 6.6 abgebildete Funktion *HelloWorld()* hinzu.

Listing 6.6 Methode Hello World(), die die Erweiterung aufruft

```
Public Function HelloWorld() As String
        Dim container As New CompositionContainer
        Dim batch As New CompositionBatch
        batch.AddPart(New TestExtension)
        batch.AddPart(Me)
        container.Compose(batch)
        Return Message
End Function
```

Betrachten wir nun, was in dieser Methode passiert.

Zuerst instanziieren wir ein Objekt vom Typ *CompositionContainer*. Dieses Objekt kann danach die MEF-Erweiterung auflösen und den gewünschten Programmcode aufrufen.

Danach instanziieren wir einen *CompositionBatch* und fügen diesem die beteiligten Klassen hinzu, das bedeutet die Klassen mit dem *Import*- und *Export*-Attribut.

Durch den Aufruf der *Compose()*-Methode des Containerobjekts wird schließlich die entsprechende Erweiterung (oder Implementierung) geladen.

Wenn Sie jetzt noch in der *Main()*-Methode das Testobjekt instanziieren und die *HelloWorld()*-Methode aufrufen, sehen Sie die in Abbildung 6.1 dargestellte Ausgabe.

Listing 6.7 Main()-Routine zum Aufruf der Programmerweiterung

```
Sub Main()
    Dim t As New Test
    Console.WriteLine(t.HelloWorld)
    Console.ReadLine()
End Sub
```

Abbildung 6.1 Ausgabe des Programms

6.2.2 MEF-Kataloge

Interessant wird die ganze Sache allerdings erst, wenn die Programmerweiterung außerhalb des Programms erfolgen kann.

Mittels sogenannter ***MEF-Catalogs*** können die Erweiterungen auch außerhalb der ursprünglichen Assembly geladen werden. MEF stellt dabei drei unterschiedliche Kataloge zur Verfügung:

- Assembly (die Erweiterung ist in einer bestimmten Assembly enthalten)
- Directory (die Erweiterung liegt innerhalb des Anwendungsverzeichnisses)

- Aggregate (eine Kombination aus Assembly und Directory)
- Type (die Erweiterung gilt für einen bestimmten Typ)

Da ich persönlich den ***Directory***-Katalog am interessantesten finde, will ich das Beispiel mit eben einem *DirectoryCatalog* fortführen.

Fügen Sie dazu Ihrer Projektmappe eine weitere Klassenbibliothek und dem neuen Projekt einen Verweis auf die Interfacebibliothek und die *System.ComponentModel.Composition*-Dll hinzu.

Listing 6.8
Implementierung der Erweiterung in einer externen Assembly

```
Imports IMEF
Imports System.ComponentModel.Composition

<Export(GetType(ITest))>
Public Class ExterneErweiterung
    Implements ITest

    Public Property Message As String _
           Implements IMEF.ITest.Message
        Get
            Return "Hello Visual Basic 10 extern aufgerufen"
        End Get
        Set(ByVal value As String)
        End Set
    End Property
End Class
```

Listing 6.8 zeigt die Erweiterung in einer externen Assembly. Der Unterschied zur vorigen Implementierung ist, dass das *Export*-Attribut einen Parameter besitzt, nämlich den Typ des zu erweiternden Interface, und nicht mehr auf die Property bezogen ist, sondern auf die gesamte Klasse. Somit weiß der *CompositionContainer*, welche Assembly er laden muss, um die entsprechende Erweiterung zu laden. Auf diese Art und Weise können Sie nicht nur einzelne Eigenschaften eines Interface erweitern, sondern sämtliche Interfacemembers. Somit ist es auch unerheblich, dass das *Import*-Attribut nicht auf Funktionen angewandt werden kann, da wir es nun auf den gesamten Interfacetyp anwenden können.

In unserer Anwendung brauchen wir bei dieser Art der Implementierung keinen Verweis auf diese neue Bibliothek zu setzen. Wir wissen schließlich auch nicht, wie ein Kunde seine kundenspezifische Bibliotheken einmal nennen will.

Vergessen Sie aber bitte nicht, die erzeugte Bibliothek manuell in das Anwendungsverzeichnis zu kopieren oder den Pfad der Build-Ausgabe einfach anzupassen.

Nun müssen wir nur noch die Klasse *Test* anpassen, damit nicht die ursprüngliche Erweiterung geladen wird, sondern die Funktionalität der gerade neu erstellten Bibliothek aufgerufen wird. Außerdem müssen wir das *Import*-Attribut jetzt auf das gesamte Interface anwenden.

Listing 6.9 zeigt die neue *Test*-Klasse.

Listing 6.9 Erweiterungsaufruf mittels eines Directorycatalog

```
Imports System.ComponentModel.Composition
Imports System.ComponentModel.Composition.Hosting
Imports System.IO
Imports System.Reflection
Imports IMEF
Public Class Test
    <Import()>
    Private _test As ITest
    Public Function HelloWorld() As String
        Dim catalog As New _
            DirectoryCatalog(Path.GetDirectoryName(Assembly.
            GetExecutingAssembly().Location))
        Dim container As New CompositionContainer(catalog)
        Dim batch As New CompositionBatch()
        batch.AddPart(Me)
        container.Compose(batch)
        Return _test.Message
    End Function
End Class
```

Dazu wurde eine private Variable *_test* vom Typ des Interface definiert und mit dem *Import*-Attribut versehen.

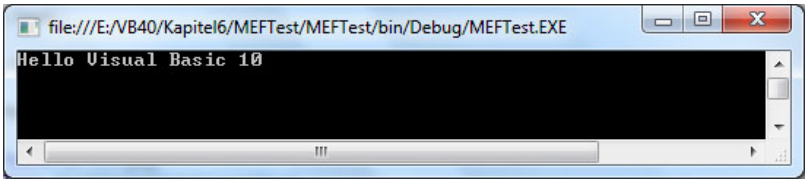
In der Methode *HelloWorld()* instanziieren wir zuerst einen *DirectoryCatalog*, der auf das aktuelle Anwendungsverzeichnis zeigt. Bei der Instanziierung unseres Containers wird der eben erzeugte *DirectoryCatalog* dem Konstruktor des Containers übergeben.

Dem *CompositionBatch* fügen wir dann noch die eigene Klasse hinzu, da sich hier die private Variable *_test* befindet, die mit dem *Import*-Attribut dekoriert wurde.

Durch den Aufruf der Methode *Compose()* wird jetzt die externe Bibliothek geladen. Falls Sie diese nicht in das Anwendungsverzeichnis kopiert haben, wird genau an dieser Stelle eine *Exception* geworfen.

Wenn Sie das Programm jetzt starten, sehen Sie, dass die Funktionalität der externen Bibliothek ausgeführt wird, was Abbildung 6.2 auch illustriert.

Abbildung 6.2 Ausgabe der Funktionalität der externen Bibliothek



Dies versteht sich nur als eine kurze Einführung in ein doch sehr mächtiges mit .NET Framework 4.0 eingeführtes Framework.

7

Windows Forms

In diesem Kapitel lernen Sie alles Wichtige über Windows Forms kennen. Anwendungen, die mit älteren Visual Basic-Versionen erstellt worden sind, unterschieden sich oft vom bekannten Microsoft Windows- oder Office-»Look and Feel«. Anwender konnten sich häufig nicht mit den Anwendungen anfreunden, da sie etwas Fremdes und Unbekanntes widerspiegelten. In Visual Studio 2010 arbeitet Microsoft gegen dieses uneinheitliche Layout. Microsoft stellt Steuerelemente zur Verfügung, die dem Endanwender bereits aus anderen Microsoft-Anwendungen bekannt sind.

7.1 Allgemein

Windows Forms oder Windows-Formulare sind zunächst einmal die Benutzeroberflächen Ihrer Anwendung. Auf ihr sind Steuerelemente platziert, die entweder Daten ausgeben oder Eingaben vom Benutzer empfangen. Auf einem Windows-Formular können Sie selbst mit Visual Studio Elemente aus der Toolbox anlegen und diese nach Ihren Wünschen anpassen. Doch erst die Programmierung der einzelnen Objekte nach Ihren Wünschen macht Ihr Programm vollständig. Formulare helfen Ihnen und dem Benutzer dabei, Informationen gezielt anzuzeigen beziehungsweise Informationen in Form von Daten an der richtigen Stelle einzugeben.

Im Laufe der Zeit wurde die grafische Benutzeroberfläche immer umfangreicher und ausgefeilter. Bereits in der ersten Visual Basic-Version konnten Benutzeroberflächen mit Steuerelementen angelegt werden. Die Möglichkeit, Windows-Formulare zu gestalten, wurde mit den darauf folgenden Visual Basic-Versionen immer umfangreicher. Heute stehen dem Entwickler eine Vielzahl von internen und externen (von Drittherstellern) Steuerelementen zur Verfügung, welche die Eingabe oder Ausgabe von Daten erheblich vereinfachen.

In diesem Kapitel werden Sie neben grundlegendem Wissen erfahren, wie Sie Ihre Windows-Formulare am besten gestalten und welche Möglichkeiten Visual Studio 2010 für eine übersichtliche Gestaltung der Steuerelemente bietet. Ebenfalls werden Sie die neuen Steuerelemente kennenlernen und

erfahren, wie Sie diese am besten in Ihre Anwendung integrieren können. Zudem gibt es Praxisbeispiele und Informationen zu typischen Problemen, mit denen viele Programmierer tagtäglich zu kämpfen haben.

Standardmäßig wird in Visual Studio 2010 eine Windows-Anwendung als Projekt angelegt. Nachdem Sie den Namen des Projekts eingegeben haben, erhalten Sie ein leeres Formular, auf dem Sie nun die Steuerelemente anlegen können.

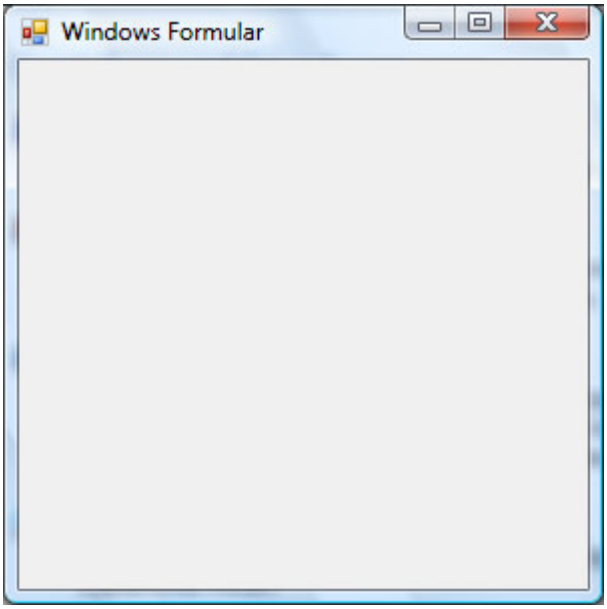


Abbildung 7.1 Neu angelegtes Windows-Formular

7.1.1 Grundlagen Windows Forms

Ein erfolgreiches Konzept schon zu den alten VB-Zeiten war Rapid Application Development (RAD). Dank einer integrierten Entwicklungsumgebung wie Visual Studio 2010 und einem großen Baukasten von Steuerelementen kann man sehr schnell aufwändige Oberflächen auf einfache Art und Weise erstellen. An diesem Konzept hat sich grundsätzlich nicht viel geändert, nur ein paar Implementierungsdetails wurden verbessert.

Eine Anwendung kann dabei aus einem oder mehreren Formularen bestehen. Sie können im Visual Studio sowohl die Entwurfsansicht als auch die Codeansicht eines Formulars anzeigen lassen. Innerhalb der Entwurfsansicht bauen Sie mit Hilfe der Toolbox die Oberfläche wie gewünscht auf, während Sie in der Codeansicht die entsprechenden Ereignishandler und sonstige Methoden implementieren können.

Neu ist jedoch, dass die Codeansicht in zwei physikalische Dateien aufgegliedert ist, eine für den eigenen geschriebenen Code und eine für den vom Designer generierten Code. Der physikalische Dateiname der Designerdatei

setzt sich wie folgt zusammen: *Formularname.Designer.vb*. Jede Aktion, die Sie in der Entwurfsansicht durchführen, wird in Programmcode umgesetzt, der in den Designercode eingetragen wird.

InitializeComponent()

Die Methode `InitializeComponent()` ist dabei das Herzstück der Designerdatei. Sie enthält alle Anweisungen, die durchgeführt werden, damit Ihr Formular zur Laufzeit so dargestellt wird, wie Sie es zusammengestellt haben. Die Methode `InitializeComponent()` wird im Konstruktor des Formulars aufgerufen.

Tipp

Wenn Sie in einem Formular einen eigenen Konstruktor implementieren, vergessen Sie bitte nicht, die Methode `InitializeComponent()` aufzurufen, andernfalls erhalten Sie ein leeres Formular.

Auch Visual Studio 2010 arbeitet diese Methode ab, um im Designer ihr Formular bereits zur Entwicklungszeit anzuzeigen.

Bestandteile einer Windows-Anwendung

Wenn Sie ein neues Projekt anlegen, sehen Sie im Projektmappenexplorer die in Abbildung 7.2 dargestellte Ansicht.

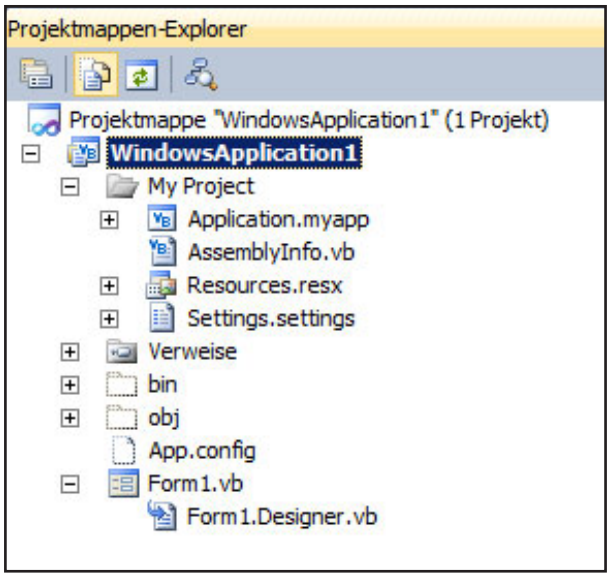


Abbildung 7.2 Projektmappenexplorer einer neuen Windows-Applikation

Hier sehen Sie recht schön, dass zu einem Formular mehrere physikalische Dateien gehören. Die dargestellte Ansicht sehen Sie im Übrigen nur, wenn Sie im Projektmappenexplorer den Button Alle Dateien anzeigen angeklickt haben.

Die **.res.x*-Datei ist noch eine zusätzliche Ressourcendatei für das Formular.

Einstellungen einer Windows-Anwendung

Durch einen Doppelklick auf den Eintrag *My Project* im Projektmappenexplorer kommen Sie zu den Einstellungsmöglichkeiten der Windows-Applikation, die in mehrere Register aufgeteilt ist.

Im Register Anwendung sehen Sie die in Abbildung 7.3 dargestellten Einstellungsmöglichkeiten.

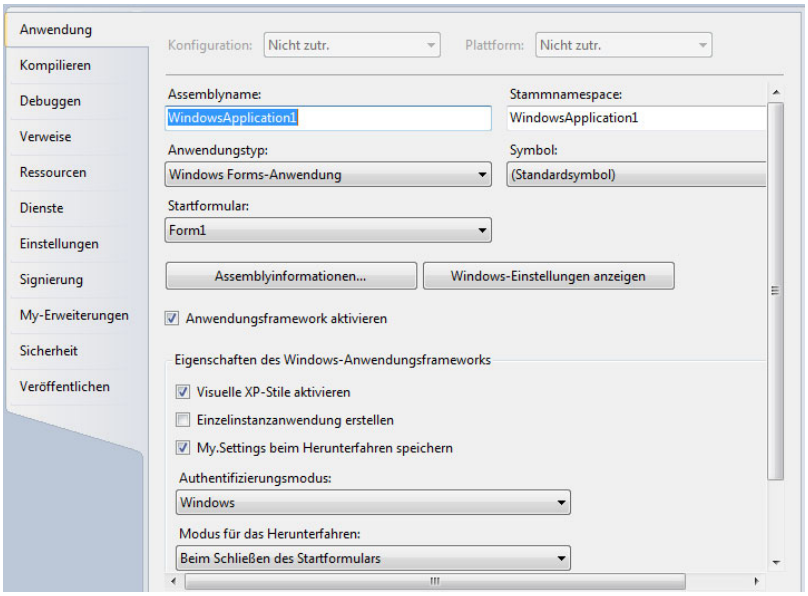


Abbildung 7.3 Einstellungen für die Anwendung

Ich möchte die wichtigsten Einstellungen hier kurz erläutern.

Unter *Assemblyname* können Sie den Namen der Exe-Datei angeben und unter *Stammnamespace* den Standardnamespace, der für die Codedateien verwendet werden soll.

Unter *Anwendungstyp* können Sie auswählen, welche Art von Anwendung kompiliert werden kann. Sie könnten hier auch alternativ Klassenbibliothek, Windows-Dienst, Konsolenanwendung oder auch Websteuerelementbibliothek auswählen. Unter *Symbol* können Sie ein Icon für die Applikation auswählen.

Unter *Startformular* wählen Sie das Formular, mit dem die Anwendung gestartet werden soll. Wollen Sie mit einer Prozedur starten, dann wählen Sie den Eintrag *Sub Main* aus, die Prozedur muss dann aber auch *Main()* heißen. Diesen Eintrag bekommen Sie jedoch erst, wenn die Markierung vor *Anwendungsframework aktivieren* deaktiviert wurde.

Visuelle XP-Stile verwenden bedeutet, dass bei gesetzter Option das eingestellte Design des Betriebssystems auf Ihr Formular angewandt wird.

In früheren Visual Basic-Versionen war es noch ein ganzes Stück Arbeit, um das mehrmalige Starten von Applikationen zu verhindern. Hier mussten Sie mit Routinen alle Prozesse im Speicher durchsuchen, um festzustellen, ob das Programm bereits geöffnet war. Hierfür hat sich Microsoft in Visual Studio 2010 etwas einfallen lassen.

Um das mehrmalige Starten zu verhindern, müssen Sie einfach den Eintrag *Einzelinstanzanwendung erstellen* aktivieren.

Bei dieser Vorgehensweise wird – bei mehrmaligem Ausführen – eine neue Instanz einfach nicht geladen. Hierbei gibt die Anwendung keine Fehlermeldung oder Ähnliches aus. Um eine eigene Fehlermeldung auszugeben, ist ein wenig Code erforderlich, den die `System.Diagnostics.Process`-Klasse für Sie bereithält.

Hierbei werden alle Prozesse im Systemspeicher durchgegangen und, falls ein und derselbe Prozess gefunden wurde, eine Meldung ausgegeben. Der Code wird in der *ApplicationEvents.vb* im `Start`-Event eingegeben und lautet wie folgt:

Listing 7.1
Prüfroutine, um doppelten Start zu vermeiden

```
Partial Friend Class MyApplication
  Private Sub MyApplication_Startup _
   (ByVal sender As Object,  _
    ByVal e As Microsoft.VisualBasic. _
      ApplicationServices.StartupEventArgs) _
    Handles Me.Startup

    Dim procName, modName As String
    modName = Process.GetCurrentProcess. _
              MainModule.ModuleName
    procName = System.IO.Path. _
       GetFileNameWithoutExtension(modName)
    'Prüfroutine
    Dim myproc() As Diagnostics.Process
    myproc = Process.GetProcessesByName(procName)
    If myproc.Length > 1 Then
       MessageBox.Show _
         ("Programm wird schon ausgeführt")
    End If
  End Sub
End Class
```

Zugriff auf die Datei *ApplicationEvents.vb* bekommen Sie durch einen Klick auf die Schaltfläche Anwendungsereignisse anzeigen.

Außerdem gibt es in diesem Register eine Option, mit der Sie festlegen können, zu welchem Zeitpunkt Ihre Applikation beendet werden soll. Hier haben Sie die Auswahl zwischen der Option, dass die Applikation beendet werden soll, wenn das Startobjekt (Ihr Formular, das beim Starten der Applikation als Erstes geladen werden soll) oder das letzte Windows-Formular in der Applikation geschlossen wird. Dies ähnelt dem Verhalten, wie es zu Visual Basic 6-Zeiten üblich war und ist über *Modus für das Herunterfahren* auszuwählen.

Fast jede größere Anwendung hat ihn, den Startbildschirm, auch **SplashScreen** genannt. Der Startbildschirm wird für eine bestimmte Zeit beim Programmstart angezeigt und enthält beispielsweise Informationen über den Autor, die Versionsnummer der Anwendung, den Titel der Anwendung sowie grafische, einprägsame Elemente. Meistens erscheint der Startbildschirm so lange, bis die Anwendung im Hintergrund vollständig geladen und einsatzbereit ist, erst dann verschwindet der Startbildschirm und lässt die Nutzung der eigentlichen Anwendung zu.

Um selbst einen Startbildschirm in Ihre Anwendung zu integrieren, fügen Sie ein neues Element vom Typ *Begrüßungsbildschirm* zu Ihrem Projekt hinzu. Dieser enthält bereits einige grafische Elemente, die von Ihnen nach eigenem Belieben angepasst werden können. Der Codedesigner hat bereits Code generiert, der Ihnen einen Teil der Arbeit abnimmt. Um den Begrüßungsbildschirm nun noch anzuzeigen, müssen Sie im Register *Anwendung* Ihrer Projekteinstellungen den Begrüßungsbildschirm unter dem Eintrag *Begrüßungsbildschirm* auswählen.

Besonders bei Anwendungen, die ohne zusätzlichen Ladevorgang auskommen, wird der Begrüßungsbildschirm leider sofort wieder geschlossen. Für die Demonstration ist ein Timer zu empfehlen, der den Begrüßungsbildschirm eine beliebige Zeit offen hält, bevor die eigentliche Anwendung startet.

Im folgenden Listing 7.2 sehen Sie Code, der für einen funktionierenden Begrüßungsbildschirm möglich wäre. Hier werden die Steuerelemente aus den Informationen der Anwendung automatisch benannt.

Listing 7.2 Beispielcode für den Begrüßungsschirm

```
Public NotInheritable Class SplashScreen1
  Private Sub SplashScreen1_Load _
    (ByVal sender As Object, _
     ByVal e As System.EventArgs) Handles Me.Load

    If My.Application.Info.Title <> "" Then
       ApplicationTitle.Text = _
         My.Application.Info.Title
    Else
      ApplicationTitle.Text = System.IO.Path. _
        GetFileNameWithoutExtension _
        (My.Application.Info.AssemblyName)
    End If
    Version.Text = System.String.Format( _
      Version.Text, _
      My.Application.Info.Version.Major, _
      My.Application.Info.Version.Minor)
   Copyright.Text = My.Application.Info.Copyright
  End Sub
End Class
```

7.1.2 Steuerelemente

In der Toolbox finden Sie eine Reihe von unterschiedlichen Standardsteuerelementen in verschiedenen Registern, wie Sie in Abbildung 7.4 sehen.

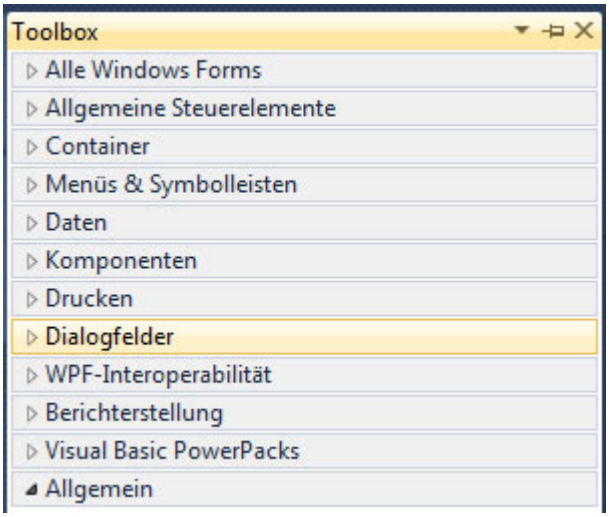


Abbildung 7.4
Toolbox-Register

Sie können Steuerelemente einfach per Drag&Drop oder durch einen Doppelklick auf ein Formular ziehen.

Betrachten wir nun aber, welche Steuerelemente in den einzelnen Registern tatsächlich vorhanden sind:

Alle Windows Forms

Hier finden Sie alle Steuerelemente innerhalb eines Registers alphabetisch sortiert.

Allgemeine Steuerelemente

In diesem Register befinden sich die am häufigsten verwendeten Steuerelemente für den alltäglichen Gebrauch. Tabelle 7.1 zeigt einen Überblick über die verfügbaren Steuerelemente:

Tabelle 7.1
Steuerelemente des Registers Allgemeine Steuerelemente

Steuerelement	Beschreibung
Button	Befehlsschaltfläche, die beim Klicken ein `Click`-Ereignis auslöst
CheckBox	Steuerelement zum Aktivieren/Deaktivieren einer Option
CheckedListBox	Liste von CheckBoxen

Tabelle 7.1 (Forts.) Steuerelemente des Registers Allgemeine Steuerelemente

Steuerelement	Beschreibung
ComboBox	Kombination aus Textfeld und einer Drop-down-Liste zur Auswahl von Werten
DateTimePicker	Steuerelement zur Auswahl eines Datums
Label	Bezeichnungsfeld
LinkLabel	Bezeichnungsfeld mit Hyperlinkfunktionalität
ListBox	Liste zur Auswahl eines oder mehrerer Werte
ListView	Liste von Elementen in unterschiedlichen Ansichtsmodi. Sie besteht aus einer Auflistung von `ListItem`-Objekten, wobei ein `ListItem`-Objekt mehrere `SubItems` besitzen kann.
MaskedTextBox	Die `MaskedTextBox` erlaubt es Ihnen, die Formatierung der Textbox schon im Vorfeld festzulegen. So können Sie sicherstellen, dass der Benutzer das Formular richtig ausfüllt und keine Eingabe tätigt, die nachher zu Verarbeitungsproblemen führen kann.
MonthCalendar	Auswahl eines Datums aus einem Monatskalender
NotifyIcon	Zeigt ein Symbol im Infobereich der Taskleiste
NumericUpDown	Steuerelement zur Eingabe von Zahlen durch inkrementelles/dekrementelles Verändern eines Werts
PictureBox	Steuerelement zur Darstellung eines Bilds
ProgressBar	Steuerelement zur Darstellung eines Fortschrittsbalkens
RadioButton	Steuerelement zur Auswahl einer einzelnen Option aus einer Gruppe von RadioButttons
RichTextBox	Steuerelement zur Eingabe von formatierten Texten im rtf-Format
TextBox	Steuerelement zur Eingabe von Texten
ToolTip	Steuerelement zur Anzeige von Informationen beim Bewegen des Mauszeigers über einem anderen Steuerelement
TreeView	Hierarchische Auflistung von Knotenelementen in einer Baumstruktur. Eine `TreeView` ist dabei eine Auflistung von `TreeNode`-Objekten.
WebBrowser	Mit dem `WebBrowser`-Steuerelement wird es Ihnen im Gegensatz zu früheren Versionen um einiges erleichtert, Ihren eigenen Webbrowser in Ihre Applikation einzubauen. Der Webbrowser beruht auf der Microsoft Internet Explorer-Technologie und unterstützt die Anzeige von allen möglichen DHTML-Seiten. Das erlaubt es Ihnen, in Ihrer eigenen Applikation interne *.html*-Dateien anzeigen zu lassen oder auf fremde Inhalte im Intra- oder Internet zu verweisen, ohne den Internet Explorer zu benutzen.

Container

In diesem Register befinden sich Steuerelemente, die wiederum andere Steuerelemente aufnehmen können. Tabelle 7.2 zeigt einen Überblick über die verfügbaren Container-Steuerelemente:

Tabelle 7.2
Steuerelemente des Registers Container

Steuerelement	Beschreibung
FlowLayoutPanel	Steuerelemente innerhalb dieses Steuerelements werden im FlowLayout angezeigt.
GroupBox	Rahmen für eine Gruppe von Steuerelementen mit Beschriftungsfunktion
Panel	Container zur Gruppierung von Steuerelementen
SplitContainer	Teilt den Anzeigebereich horizontal oder vertikal in zwei Bereiche, die der Benutzer während der Laufzeit in der Größe verändern kann
TabControl	Auflistung von Registerkarten, die wiederum andere Steuerelemente enthalten können
TableLayoutPanel	Steuerelemente innerhalb dieses Steuerelements werden im Tabellenformat angezeigt.

Menüs & Symbolleisten

In diesem Register befinden sich Menüs und Symbolleisten. Tabelle 7.3 zeigt einen Überblick über die verfügbaren Menüs und Symbolleisten:

Tabelle 7.3
Steuerelemente des Registers Menüs & Symbolleisten

Steuerelement	Beschreibung
ContextMenuStrip	Zeigt bei einem Rechtsklick auf ein Steuerelement ein Kontextmenü an
MenuStrip	Benutzermenü mit zusätzlichen Features, so wie Sie es von Microsoft Office her kennen
StatusStrip	Statusleiste zur Anzeige von Informationen oder anderen Statusmeldungen
ToolStrip	Symbolleiste mit unterschiedlichen Layoutoptionen
ToolStripContainer	Container für MenuStrip-, StatusStrip- und ToolStrip-Elementen

Daten

In diesem Register befinden sich Steuerelemente und Komponenten für den Datenzugriff. Tabelle 7.4 zeigt einen Überblick über die verfügbaren Datensteuerelemente:

Tabelle 7.4
Steuerelemente des Registers Daten

Steuerelement	Beschreibung
Chart	Steuerelement zum Anzeigen von Diagrammen
BindingNavigator	Steuerelement zur Steuerung der Navigation in gebundenen Daten; stellt Blätterfunktionalität zur Verfügung

Tabelle 7.4 (Forts.) Steuerelemente des Registers Daten

Steuerelement	Beschreibung
BindingSource	Steuerelement zum Verbinden von anderen Steuerelementen mit Daten
DataGridView	Steuerelement zum Anzeigen und Bearbeiten von Daten in einer tabellarischen Darstellung
DataSet	Lokaler Datenspeicher bei verbindungslosen Datenbankanwendungen

Komponenten

In diesem Register befinden sich Steuerelemente, die während der Laufzeit nicht auf dem Formular erscheinen, sondern im Hintergrund arbeiten. Steuerelemente aus diesem Register werden im Komponentenfach dargestellt. Tabelle 7.5 zeigt einen Überblick über die verfügbaren Komponenten:

Tabelle 7.5 Steuerelemente des Registers Komponenten

Steuerelement	Beschreibung
BackgroundWorker	Diese Komponente stellt einen eigenen Thread zur Hintergrundverarbeitung von lang laufenden Operationen zur Verfügung.
DirectoryEntry	Eintrag im ActiveDirectory
DirectorySearcher	Kapselt den Zugriff auf das ActiveDirectory
ErrorProvider	Ermöglicht die Darstellung von Informationen zu fehlerhaften Benutzereingaben
EventLog	Diese Komponente bietet Zugriff auf das Ereignisprotokoll.
FileSystemWatcher	Komponente zur Überwachung eines Verzeichnisses auf Änderungen, Neuanlagen oder Löschungen in diesem Verzeichnis oder einem Unterverzeichnis
HelpProvider	Ermöglicht die Darstellung von zusätzlichen Hilfeinformationen für bestimmte Steuerelemente
ImageList	Speichert Bilder, die in anderen Steuerelementen wieder verwendet werden können
MessageQueue	Diese Komponente bietet Zugriff auf die Nachrichtenwarteschlange des Systems.
PerformanceCounter	Diese Komponente bietet Zugriff auf unterschiedliche PerformanceCounter.
Process	Diese Komponente kapselt den Zugriff auf einen Prozess.
SerialPort	Ermöglicht die Kommunikation über die serielle Schnittstelle
ServiceController	Diese Komponente kapselt den Zugriff auf einen Dienst.
Timer	Steuerelement, das innerhalb von bestimmten Intervallen ein `Tick`-Ereignis auslöst

Drucken

In diesem Register befinden sich Steuerelemente, um Druckfunktionalität zu implementieren. Tabelle 7.6 zeigt einen Überblick über die verfügbaren Drucksteuerelemente:

Tabelle 7.6
Steuerelemente des Registers Drucken

Steuerelement	Beschreibung
PageSetupDialog	Dialogfeld, um benutzerdefinierte Seiteneinstellungen zu ändern
PrintDialog	Dialogfeld zur Auswahl eines Druckers oder von Druckoptionen
PrintDocument	Objekt, das Ausgaben an einen Drucker sendet. Das Layout wird dabei nicht mit einem Designer, sondern per Programmcode mit Grafikmethoden erstellt.
PrintPreviewControl	Steuerelement zur Darstellung einer Druckvorschau. Das anzuzeigende Dokument ist vom Typ `PrintDocument`.
PrintPreviewDialog	Dialog zur Darstellung einer Druckvorschau. Enthält im Gegensatz zum `PrintPreviewControl` weitere Schaltflächen und Dialogfelder.

Dialogfelder

In diesem Register befinden sich die bekannten Windows-Standarddialoge. Tabelle 7.7 zeigt einen Überblick über die verfügbaren Standarddialoge:

Tabelle 7.7
Steuerelemente des Registers Dialogfelder

Steuerelement	Beschreibung
ColorDialog	Standarddialog zur Auswahl von Farben
FolderBrowserDialog	Standarddialog zur Auswahl eines Ordners
FontDialog	Standarddialog zur Auswahl einer Schriftart
OpenFileDialog	Datei-Öffnen-Standarddialog
SaveFileDialog	Datei-Speichern-Standarddialog

WPF-Interoperabilität

Besteht aus einem einzigen Steuerelement `ElementHost`, um Interoperabilität mit der modernen WPF-Technologie zu gewährleisten.

Berichterstellung

In diesem Register befindet sich ein zur Erstellung von Berichten verwendetes Steuerelement `Microsoft Report Viewer`.

Achtung

Dieses Steuerelement ist zwar im Funktionsumfang von Visual Studio 2010 enthalten, es ist aber nicht Bestandteil der .NET 4.0-Laufzeitumgebung. Für den Einsatz dieses Steuerelements benötigen Sie eine zusätzliche Installation an den Clients.

Visual Basic PowerPacks

In diesem Register befinden sich für Visual Basic bekannte Steuerelemente. Tabelle 7.8 zeigt einen Überblick über die verfügbaren Visual Basic PowerPacks-Steuerelemente:

Tabelle 7.8 Steuerelemente des Registers Visual Basic PowerPacks

Steuerelement	Beschreibung
PrintForm	Zum einfachen Drucken einer Windows Form als Bericht
LineShape	Zum einfachen Zeichnen einer Linie
OvalShape	Zum einfachen Zeichnen eines ovalen Grafikobjekts
RectangleShape	Zum einfachen Zeichnen eines rechteckigen Grafikobjekts
DataRepeater	Zum Anzeigen von Daten in einem anpassbaren Listenformat

Allgemein

Leeres Register. In diesem Register können auch Text oder Codeblöcke hinzugefügt werden, die danach in die Codeansicht gezogen werden können.

In der nachfolgenden Tabelle 7.9 will ich Ihnen noch einen Überblick über Eigenschaften bieten, die Sie bei vielen der gerade aufgelisteten Steuerelemente wiederfinden.

Tabelle 7.9 Eigenschaften von Steuerelementen

Eigenschaft	Beschreibung
(Name)	Name, unter dem das Steuerelement per Code angesprochen wird. Jedes Steuerelement hat eine formularweit eindeutige `Name`-Eigenschaft.
AccessibleDescription, AccessibleName, AccessibleRole	Eigenschaften zur Integration von Hilfen für Blinde und Sehbehinderte
AllowDrop	Bestimmt, ob das Steuerelement Drag&Drop unterstützt
Anchor	Über diese Eigenschaft kann eingestellt werden, zu welchen Rändern seines Containers das Steuerelement bei Größenänderungen zur Laufzeit den Abstand beibehält. Ein Steuerelement kann an einer beliebigen Kombination von Rändern verankert werden; beim Verankern an gegenüberliegenden Kanten verändert das Control seine Größe.
BackColor	Bestimmt die Hintergrundfarbe eines Steuerelements. Bei Containern wird diese Eigenschaft vererbt.

Tabelle 7.9 (Forts.) Eigenschaften von Steuerelementen

Eigenschaft	Beschreibung
CausesValidation	Gibt an, ob beim Aktivieren des Steuerelements das `Validating`-Ereignis des vorher aktiven Steuerelements ausgelöst wird
ContextMenuStrip	Über diese Eigenschaft kann einem Steuerelement ein Kontextmenü zugeordnet werden.
Controls	Enthält bei Container-Steuerelementen eine Auflistung der untergeordneten Steuerelemente
Cursor	Bestimmt den Mauszeiger eines Steuerelements. Bei Containern wird diese Eigenschaft vererbt.
Dock	Gibt an, an welchem Rand seines Containers das Steuerelement anliegt
Enabled	Bestimmt, ob ein Steuerelement aktiv/inaktiv ist. Bei Containern wird diese Eigenschaft vererbt.
Font	Gibt die Schriftart eines Steuerelements an. Bei Containern wird diese Eigenschaft vererbt.
ForeColor	Bestimmt die Vordergrundfarbe eines Steuerelements. Bei Containern wird diese Eigenschaft vererbt.
Location	Bestimmt die linke obere Ecke eines Steuerelements
Locked	Bestimmt, ob das Steuerelement zur Designzeit verschoben werden kann
Modifiers	Bestimmt den Zugriffsmodifizierer des Steuerelements. Bei Visual Basic standardmäßig `Friend`.
RightToLeft	Gibt an, ob das Steuerelement bei entsprechenden Kulturen seinen Text von rechts nach links ausgibt
Size	Bestimmt die Höhe und Breite des Steuerelements
TabStop	Gibt an, ob das Steuerelement mit der Tabulatortaste erreicht werden kann
Tag	Ermöglicht es, das Steuerelement mit beliebigen Daten zu verbinden. Diese Eigenschaft ist vom Typ `Object` und somit kann jegliche Art von Objekten an das `Tag` angebunden werden.
Visible	Gibt an, ob das Steuerelement zur Laufzeit sichtbar ist

In der nachfolgenden Tabelle 7.10 will ich Ihnen noch einen Überblick über Ereignisse bieten, die Sie bei vielen der gerade aufgelisteten Steuerelemente wiederfinden.

Tabelle 7.10 Ereignisse von Steuerelementen

Ereignis	Beschreibung
Click	Wird ausgelöst, wenn auf das Steuerelement geklickt wird
DoubleClick	Wird ausgelöst, wenn doppelt auf das Steuerelement geklickt wird
DragDrop, DragEnter, DragLeave, DragOver	Ereignisse, die beim Drag&Drop-Vorgang ausgelöst werden, um Rückmeldung an den Benutzer zu geben
Enter	Wird ausgelöst, wenn das Steuerelement zum aktiven Steuerelement des Formulars wird
KeyDown, KeyPress, KeyUp	Diese Ereignisse werden durch Tastatureingaben ausgelöst.
Leave	Wird ausgelöst, wenn das Steuerelement den Fokus verliert
MouseClick	Dieses Ereignis wird nur beim Klicken mit der Maus ausgelöst, das `Click`-Ereignis kann dagegen auch durch die EINGABE-Taste ausgelöst werden.
MouseDown, MouseEnter, MouseLeave, MouseMove, MouseUp	Diese Ereignisse werden bei entsprechenden Mausaktionen ausgelöst.
Paint	Dieses Ereignis wird ausgelöst, wenn das Steuerelement sich neu zeichnet.
Resize	Wird ausgelöst, wenn die Größe des Steuerelements geändert wird
Validating	Wird ausgelöst, während der Inhalt des Steuerelements validiert wird
Validated	Wird ausgelöst, nachdem der Inhalt des Steuerelements erfolgreich validiert wurde

7.1.3 Eingabevalidierung

Nachdem Sie jetzt gelesen haben, welche unterschiedlichen Steuerelemente es gibt, und einige Eigenschaften sowie Ereignisse kennengelernt haben, geht es nun darum, wie diese auch richtig eingesetzt werden.

Eine wesentliche Aufgabe, die in fast jeder Applikation benötigt wird, ist die Validierung von Benutzereingaben, um sicherzustellen, dass auch nur plausible Daten weiterverarbeitet werden.

Dazu eignet sich in der Regel das `Validating`-Ereignis eines Steuerelements. Dieses Ereignis tritt nach dem `Leave`-Ereignis auf, jedoch kann der Verlust des Fokus bei einer fehlerhaften Eingabe dadurch verhindert werden, dass die Eigenschaft `Cancel` der Klasse `CancelEventArgs` auf `True` gesetzt wird. Somit kann das Steuerelement nicht verlassen werden, solange keine vernünftige Eingabe erfolgt ist.

In vielen Fällen ist es aber nötig, einen Fluchtweg offen zu lassen, um zum Beispiel ein Formular ohne Weiterverarbeitung der Daten wieder zu verlassen. Setzen Sie einfach die `CausesValidation`-Eigenschaft eines Abbrechen-Buttons auf `False` und schon wird das `Validating`-Ereignis des zu überprüfenden Steuerelements nicht ausgelöst.

Wenn Sie den Verlust des Fokus verhindern, dann sollten Sie dem Benutzer auch eine Rückmeldung geben, dass eine falsche Eingabe erfolgte. Eine `MessageBox` erweist sich dabei auf Dauer sehr schnell als nervig, eine schöne Alternative ist ein Hinweis mit Hilfe eines `ErrorProvider`.

Im folgenden Listing 7.3 sehen Sie den kompletten Code für die Überprüfung des Inhalts einer `TextBox` und den Einsatz eines `ErrorProvider`.

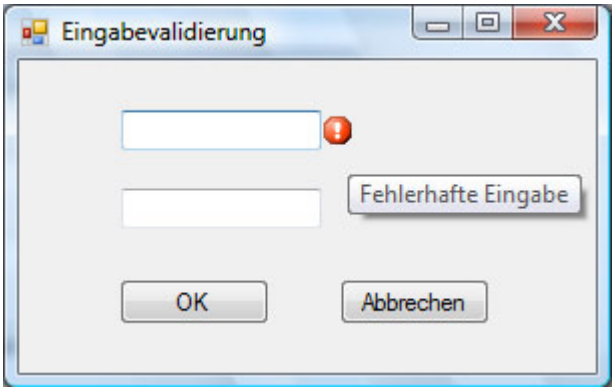
Listing 7.3
Validierung einer TextBoxeingabe

```
Public Class Form1

  Private Sub TextBox1_Validating _
    (ByVal sender As System.Object, _
     ByVal e As _
         System.ComponentModel.CancelEventArgs) _
     Handles TextBox1.Validating

    If TextBox1.Text.Length < 5 Then
      e.Cancel = True
      ErrorProvider1.SetError(TextBox1, _
           "Fehlerhafte Eingabe")
    Else
      ErrorProvider1.SetError(TextBox1, "")
    End If
  End Sub

  Private Sub btnCancel_Click _
    (ByVal sender As System.Object, _
     ByVal e As System.EventArgs) _
     Handles btnCancel.Click

     Me.Close()
  End Sub

  Private Sub Form1_FormClosing _
    (ByVal sender As System.Object, _
     ByVal e As FormClosingEventArgs) _
     Handles MyBase.FormClosing

     e.Cancel = False
  End Sub
End Class
```

Sie sehen in Listing 7.3, dass der Text des `ErrorProvider` bei einer korrekten Eingabe auf einen leeren String gesetzt wird. Damit wird er wieder deaktiviert. Außerdem setzen wir in der Methode `Form1_FormClosing()` die Eigenschaft `e.Cancel` wieder auf `False`, damit das Formular auch wieder verlassen werden kann. Tun Sie das nicht, stellt sich die Abbrechen-Schaltfläche dar, als hätte sie keine Funktionalität.

Abbildung 7.5 zeigt die kleine Anwendung und die Darstellung des `ErrorProvider` rechts neben der `TextBox`.

Abbildung 7.5
Eingabevalidierung



Tipp

Auch wenn Sie sich an dieser Stelle schon viel Mühe gemacht haben, um die Überprüfung der Eingabe auf Steuerelementebene durchzuführen, sollten Sie zusätzlich eine formularweite Plausibilitätsüberprüfung implementieren. Erhält nämlich ein Steuerelement niemals den Fokus, wird auch kein `Validating`-Ereignis ausgeführt. Folglich sollten vor der Weiterverarbeitung der Daten stets noch globale Überprüfungen stattfinden.

7.2 Formulare

Nachdem wir jetzt einiges über Steuerelemente gelesen haben, kommen wir nochmals zurück zu den Grundcontainern, den Formularen.

Ein Formular ist in Visual Basic nichts anderes als eine Klasse, die dabei von einer Basisklasse `Form` erbt. Somit ist der Grundaufbau des Formulars bereits über die Vererbung vorgegeben.

Im Folgenden betrachten wir ein paar Besonderheiten von Windows Forms.

7.2.1 Transparenz

Dank der allgemeinen Transparenzunterstützung ist es möglich, ein Formular transparent zu gestalten. Im Formular gibt es eine Eigenschaft `Opacity`. Während der Entwicklung wird diese in einer Prozentzahl zwischen

0 und 100 angegeben, wobei 0% die vollständige Transparenz und 100 % keine Transparenz bedeutet. Während der Laufzeit kann dieser Wert ebenfalls verändert werden. Hier aber nicht mit einer Angabe zwischen 1 und 100, sondern zwischen 0.0 und 1.0, wobei 0.0 wiederum für die vollständige Transparenz steht. Im folgenden Beispiel *WinBasic* wird das Steuerelement `NumericUpDown` verwendet, um die Transparenz während der Laufzeit anzupassen. Hierzu werden auf dem Formular nur dieses zusätzliche Steuerelement sowie der Code aus Listing 7.4 benötigt:

Listing 7.4
Anpassen der Transparenz

```
Private Sub NumericUpDown1_ValueChanged _
  (ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles NumericUpDown1.ValueChanged

   Me.Opacity = Me.NumericUpDown1.Value
End Sub
```

Abbildung 7.6 zeigt das transparente Formular.

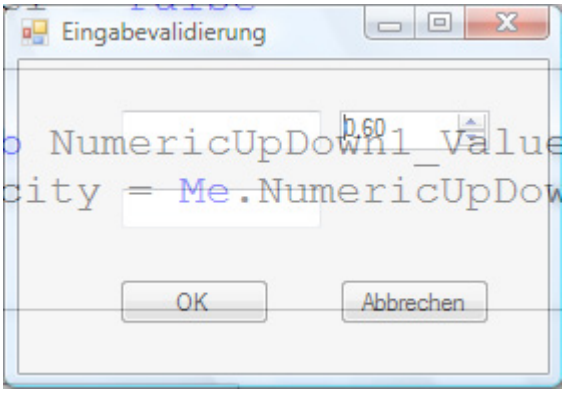


Abbildung 7.6
Änderung der Transparenz zur Laufzeit

Neben dem gesamten Formular ist es auch möglich, bestimmte Teile des Formulars transparent zu gestalten. Hierzu dient die `TransparencyKey`-Eigenschaft. Mit dieser Eigenschaft setzen Sie fest, welche Farben des Formulars komplett transparent dargestellt werden sollen. So ist es zum Beispiel möglich, den grauen Hintergrund transparent zu verstecken, ein Element in der Mitte aber besonders hervorzuheben. Das klingt zunächst wie Spielerei, kann jedoch für eine klare Programmführung äußerst interessant sein. So können Sie zum Beispiel vom Benutzer auszufüllende Elemente hervorheben oder eigentlich überladene Formulare übersichtlicher gestalten.

Eine weitere Möglichkeit besteht darin, als Hintergrundbild (`BackGroundImage`) eines Formulars ein Bild mit einer transparenten Farbe einzubinden. Wenn Sie danach den `TransparencyKey` auf die entsprechende Farbe und die Eigenschaft `FormBorderStyle` auf `None` setzen, haben Sie plötzlich kein eckiges Formular mehr, wie Sie in Abbildung 7.7 sehen können.

Abbildung 7.7
Nicht eckiges Formular

7.2.2 Tabulatorreihenfolge innerhalb eines Formulars

Sie sollten auch immer darauf achten, dass Sie in Ihren Programmen den Benutzer von links oben nach rechts unten durch ein Formular führen. Dies können Sie mit der Aktivierreihenfolge in Ihrem Formular definieren. Das ist die Reihenfolge, mit der ein Benutzer die einzelnen Elemente über die ⇆-Taste erreichen kann.

Über das Menü Ansicht – Aktivierreihenfolge können Sie die Aktivierreihenfolge Ihres Formulars einblenden bzw. wieder ausblenden, wie Sie in Abbildung 7.8 sehen können.

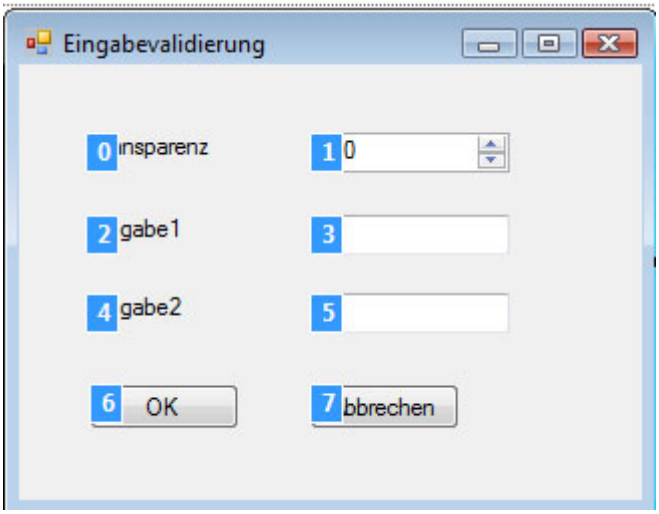


Abbildung 7.8
Aktivierreihenfolge

Die Zahl, die Sie jeweils links neben den integrierten Komponenten sehen, gibt die Aktivierreihenfolge wieder. Sie können sie per Mausklick ändern. Die neue Aktivierreihenfolge wird dann sofort übernommen und in die Eigenschaft `TabIndex` des jeweiligen Steuerelements eingetragen.

Wenn Sie sich Abbildung 7.8 betrachten, werden Sie sich vielleicht die Frage stellen, warum denn auch Labels einen `TabIndex` besitzen, obwohl sie doch nie den Fokus bekommen können. Sie können aber für Labels durch ein `&` in der Texteigenschaft den Buchstaben, der dem `&` folgt, als Shortcut definieren. Wenn nun ein Benutzer diesen Shortcut (Alt + Zeichen) wählt, würde der Fokus auf dieses `Label` gesetzt werden. Jedoch kann das `Label` den Fokus nicht erhalten, also wird in der Aktivierreihenfolge um eins weitergegangen. Sie sollten also den `TabIndex` einer `TextBox` immer um eins höher setzen als den `TabIndex` des zugehörigen `Label`.

Tipp

In vielen Formularen gibt es auch sogenannte Standardschaltflächen, die Sie mit ↵ aktivieren können, und auch Schaltflächen, die Sie mit Esc auslösen können. Über die Eigenschaften eines Formulars können Sie diese Schaltflächen mittels `AcceptButton` und `CancelButton` festlegen.

7.2.3 Kommunikation zwischen Formularen

Die wenigsten Windows-Anwendungen bestehen nur aus einem einzigen Formular. In dem Moment, wo Sie mehrere Formulare in Ihrer Anwendung haben, besteht die Notwendigkeit, Daten zwischen Formularen auszutauschen.

Sie können ein zweites Formular mit zwei unterschiedlichen Methoden aufrufen, als modales und als nicht modales Formular.

Bei einem modalen Aufruf, mittels der Methode `ShowDialog()`, spricht man sehr häufig auch von Dialogen. Das bedeutet, dass auf das aufrufende Formular erst wieder zugegriffen werden kann, wenn der Dialog beendet wird. Der Programmcode im aufrufenden Formular bleibt an der Zeile `ShowDialog()` stehen, bis der aktuelle Dialog geschlossen wird. Bei einem nicht modalen Aufruf, mittels der Methode `Show()`, kann gleichzeitig an beiden Formularen gearbeitet werden.

Nicht modale Formulare

Um ein Formular überhaupt aufzurufen, sollten Sie es davor instanziieren. Die folgenden Zeilen zeigen den Programmcode zum Aufruf eines nicht modalen Formulars.

```
Dim frm As New frmNeuesFormular
frm.Show()
```

Um zwischen den einzelnen Formularen zu kommunizieren, müssen Sie lediglich öffentlich zugängliche Methoden anlegen, die dann über die Instanzvariable aufgerufen werden können.

Dialoge – modale Formulare

Ein bisschen anders sieht es beim Aufruf von modalen Dialogen mit der Methode `ShowDialog()` aus. Dabei besitzt die Methode `ShowDialog()` einen Rückgabewert vom Typ `DialogResult`.

Schaltflächen besitzen dabei eine Eigenschaft `DialogResult`, die Sie auf einen Enumerationswert setzen können. Wenn Sie nun in einem Dialog eine Schaltfläche anklicken, bei der ein `DialogResult`-Wert angegeben wurde, so wird das Formular geschlossen und der Wert als Rückgabewert der Methode `ShowDialog()` gesetzt, ohne eine eigene Zeile Programmcode dafür zu schreiben.

Dadurch, dass die Steuerelemente in Visual Basic 10 als `Friend` definiert sind, kann man in Abhängigkeit vom Rückgabewert auf jegliches Steuerelement im Dialog über die instanziierte Formularvariable zugreifen, wie Sie in Listing 7.5 sehen können.

Listing 7.5 Aufruf eines Dialogs

```
Dim frm As New frmNeuesFormular
If frm.ShowDialog() = DialogResult.OK Then
    'Über frm auf die Elemente zugreifen
End If
```

Formularauflistung

Eventuell standen Sie bereits in früheren Versionen von Visual Basic .NET vor dem Problem, dass Sie alle geöffneten Formulare anzeigen mussten. Das war ohne eigenen Code, der jedes der Formulare protokolliert, kaum möglich. Mittlerweile ist dies jedoch sehr einfach zu bewerkstelligen. Der Code in Listing 7.6 demonstriert eine Schleife, die alle offenen Fenster auflistet und deren Überschrift ausgibt:

Listing 7.6 Formularauflistung

```
Private Sub offeneFormulareZaehlen()
  For Each frm As Form In _
    My.Application.OpenForms

    MessageBox.Show(frm.Text)
  Next
End Sub
```

7.2.4 Lokalisierung

Entwickeln Sie mehrsprachige Anwendungen? In früheren Versionen von Visual Studio .NET war es noch relativ kompliziert, Steuerelemente und Formulare in verschiedene Sprachen zu übersetzen. Hier hat Microsoft nachgelegt und die Lokalisierungsfunktionalität erweitert.

In der folgenden Abbildung 7.9 sehen Sie unser Beispielformular mit deutschen Beschriftungen und der Ausgabe des aktuellen Datums in einem deutschen Format.

Ich habe gemäß der Spracheinstellung *en-US* ins Englische übersetzt und die beiden Buttons vertauscht.

Abbildung 7.9
Formular mit deutscher Beschriftung

Nachdem Sie die Anwendung erneut starten, wird auf die vom System hinterlegte Sprache (ich gehe mal von Deutsch aus) zurückgegriffen. Um die Sprache während der Laufzeit zu ändern, müssen Sie die Eigenschaft `CurrentUICulture` des aktuellen Threads ändern. Diese Änderung sollte unbedingt vor dem Aufruf der Methode `InitializeComponent()` erfolgen.

Listing 7.7 zeigt, wie Sie den Aufruf in den Konstruktor des Formulars mit einbauen.

Listing 7.7
Aktuelle Sprache ändern

```
Sub New()
  'Sprache ändern
  System.Threading.Thread.CurrentThread. _
    CurrentUICulture = New _
    System.Globalization.CultureInfo("en-US")
  InitializeComponent()
End Sub
```

Wenn Sie das Formular jetzt starten, sehen Sie, dass die englische Version am Bildschirm angezeigt wird. Allerdings erscheint das Datum noch immer im deutschen Format.

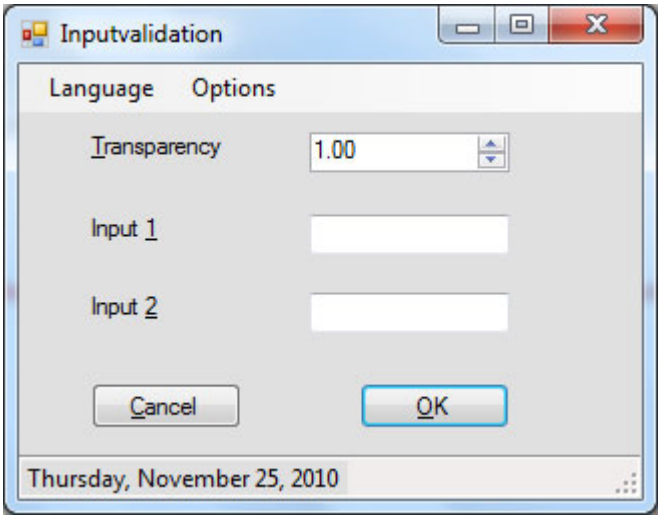
Mit der Eigenschaft `CurrentUICulture` werden nur die Anpassungen ausgewertet, die in den Ressourcendateien hinterlegt sind. Wenn Sie auch die Kulturanpassung für Datums-, Zahlen- und Währungsformate durchführen wollen, müssen Sie die `CurrentCulture` ebenfalls umstellen.

Sie müssen also den Konstruktor noch um folgende Zeile erweitern (auch wieder vor dem `InitializeComponent()`-Aufruf.

```
System.Threading.Thread.CurrentThread. _
  CurrentCulture = New _
  System.Globalization.CultureInfo("en-US")
```

Wenn Sie danach die Anwendung wieder starten, sehen Sie das Formular wie in Abbildung 7.10 dargestellt.

Abbildung 7.10
Lokalisiertes Formular



Ohne jetzt kleinlich wirken zu wollen, haben wir noch ein weiteres Problem. In dem Moment, wo unser `ErrorProvider` eine Falscheingabe registriert, gibt er als Text noch immer den deutschen Text *Fehlerhafte Eingabe* aus.

Das ist auch nicht weiter verwunderlich, denn diesen Text haben wird per Programmcode gesetzt.

```
ErrorProvider1.SetError(TextBox1, "Fehlerhafte Eingabe")
```

Ein ähnliches Problem würde auch bei der Ausgabe von Nachrichten mittels einer `MessageBox` auftreten.

Die Lösung für dieses Problem sind nicht Ressourcendateien, die an Formularen hängen, sondern globale Ressourcendateien.

Mittels Projekt – Neues Element hinzufügen... können Sie im folgenden Dialog (siehe Abbildung 7.11) eine Ressourcendatei zu Ihrem Projekt hinzufügen. Die Ressourcendatei habe ich in diesem Fall *AllgemeineTexte.resx* genannt.

Anschließend können Sie für jeden Text, den Sie lokalisieren wollen, einen Schlüssel in der Spalte *Name* vergeben und als *Wert* den tatsächlichen Text eintragen, wie Sie in Abbildung 7.12 sehen.

Nachdem Sie alle Einträge erfasst haben, speichern Sie die Datei. Kopieren Sie anschließend die Datei und fügen Sie die Kopie dem Projekt hinzu. Benennen Sie diese in *AllgemeineTexte.en-US.resx* um. Der Name der lokalisierten globalen Ressourcendatei ist der Name der Standardressourcendatei, gefolgt von einem Punkt und der Kulturinformation, gefolgt von der Dateiendung *.resx*.

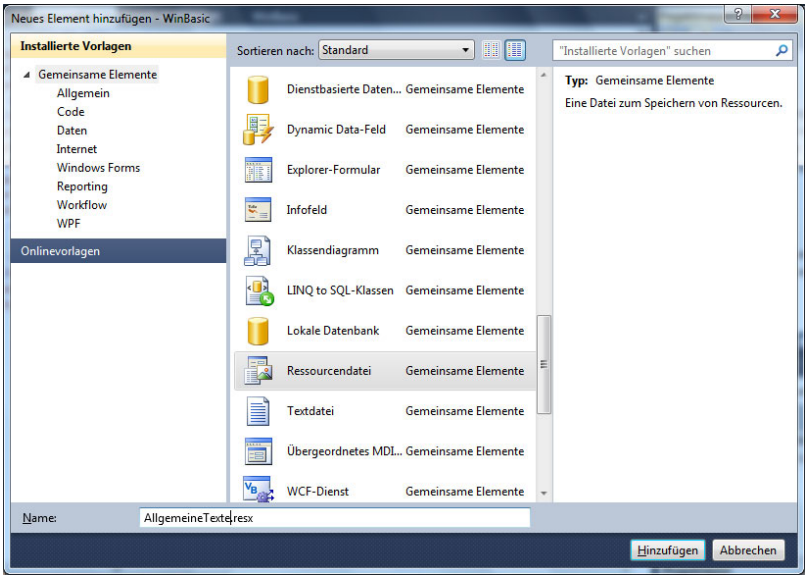


Abbildung 7.11
Ressourcendatei hinzufügen

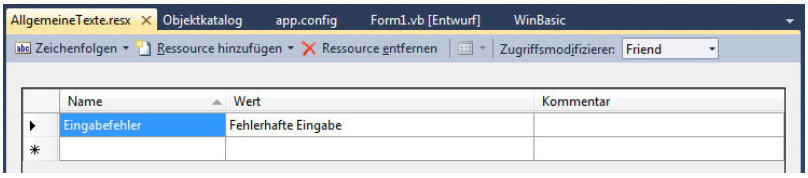


Abbildung 7.12
Erfassen der lokalisierten Texte

Jetzt können Sie die entsprechende Übersetzung in der lokalisierten Ressourcendatei durchführen, wie Sie in Abbildung 7.13 sehen.

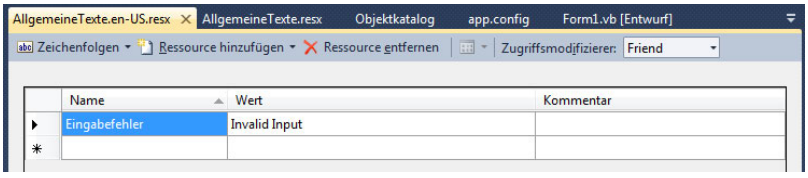


Abbildung 7.13
Lokalisierte AllgemeineTexte.en-US.resx

Bleibt zum Schluss nur noch die Frage, welche Änderungen im Programmcode nötig sind, um den anzuzeigenden Text aus der Ressource auszulesen.

Visual Studio 2010 erzeugt eine generische Klasse aus der Ressourcendatei und die Einträge in der Spalte *Name* sind die Namen von Properties dieser generischen Klasse, die den Wert aus der Ressourcendatei der gerade aktuellen Kultur ausliest.

Sie müssen also nur die Zeile

```
ErrorProvider1.SetError(TextBox1, "Fehlerhafte Eingabe")
```

durch folgende Anweisung ersetzen:

```
ErrorProvider1.SetError(TextBox1, _
     My.Resources.AllgemeineTexte.Eingabefehler)
```

7.2.5 Formularvererbung

Jedes Formular erbt von einer Basisklasse `System.Windows.Forms.Form`. Innerhalb dieser Klasse sind alle grundsätzlichen Formulareigenschaften, -ereignisse und -methoden implementiert.

Sie können aber auch eine eigene Formularbibliothek aufbauen, um Vorlagen für Dialoge, Eingabemasken und sonstige Standardformulare zu erstellen. Formularvererbung gewährleistet ein einheitliches Layout und einen einheitlichen Formularaufbau für gleichartige Formulartypen und Sie müssen bestimmte Elemente nicht jedes Mal neu auf die Formulare ziehen, sondern können einfach von diesen Vorlagen erben. Ändern Sie die Vorlage, so schlagen die Änderungen aufgrund der Vererbung auch auf die von dieser Vorlage abgeleiteten Formulare durch.

So könnten Sie zum Beispiel ein Formular mit Ihrem Firmenlogo und einer `StatusStrip` anlegen und neue Formulare anlegen, die von diesem Formular erben.

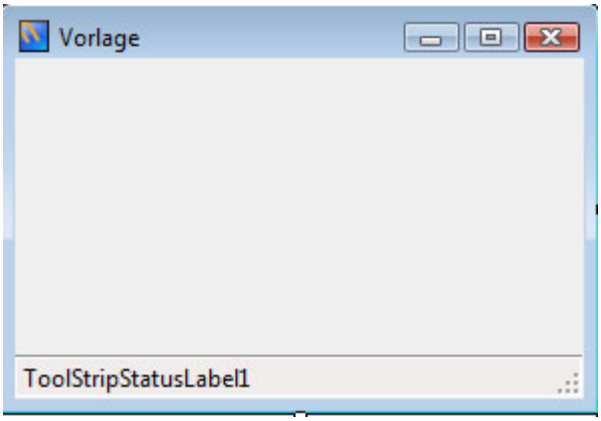


Abbildung 7.14
Formularvorlage

Außerdem habe ich noch folgenden Programmcode hinzugefügt

```
Private Sub Vorlage_Load(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load
   ToolStripStatusLabel1.Text = Now.ToLongDateString
End Sub
```

um in der `StatusStrip` das aktuelle Datum anzuzeigen.

Um nun ein neues Formular anzulegen, das von dieser Vorlage erbt, können Sie mit Projekt – Windows Form hinzufügen... ein neues geerbtes Formular hinzufügen, wie in Abbildung 7.15 dargestellt. Wählen Sie dazu unter Installierte Vorlagen – Gemeinsame Elemente den Eintrag Windows Forms aus.

Im nächsten Dialog (Abbildung 7.16) können Sie das Basisformular, von dem geerbt werden soll, auswählen. Dabei werden alle Formulare des aktuellen Projekts aufgelistet. Liegt die Vorlage in einer anderen Assembly, so können Sie mit der Schaltfläche DURCHSUCHEN... die entsprechende Assembly im Dateisystem auswählen. Der Liste werden dann die Formulare der entsprechenden Assembly hinzugefügt. Im Beispiel ist die Vorlage in der aktuellen Assembly bereits enthalten und durch Auswahl von *Vorlage* wird ein neues Formular angelegt, welches von dem Formular *Vorlage* erbt (Abbildung 7.17).

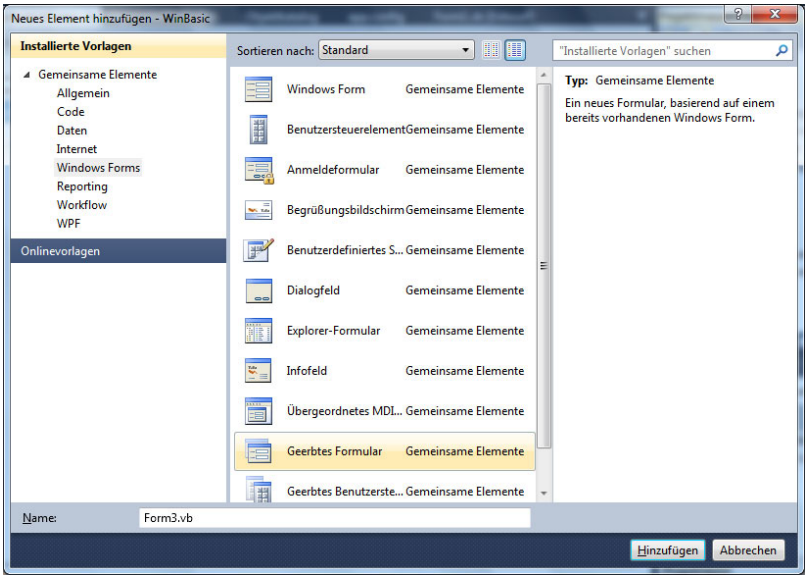


Abbildung 7.15
Hinzufügen eines geerbten Formulars

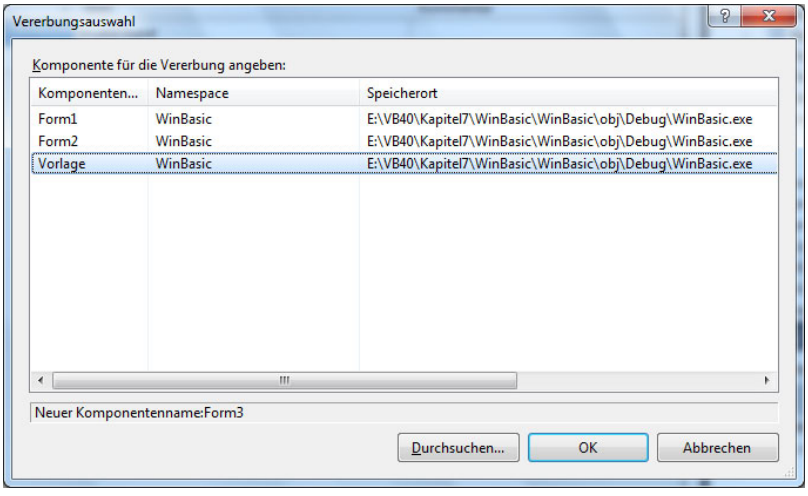


Abbildung 7.16
Auswahl des Basisformulars für die Vererbung

Achtung

Sollte die gewünschte Vorlage nicht in der Liste enthalten sein, dann übersetzen Sie die Assembly noch ein Mal, denn an dieser Stelle werden die Komponenten nicht aus dem Code, sondern aus der Assembly ausgelesen.

Abbildung 7.17
Neues Formular, das von der Vorlage erbt

In Abbildung 7.17 sehen Sie, dass in der `StatusStrip` das Datum bereits eingetragen ist. Der Programmcode, der das Datum setzt, wurde bereits ausgeführt, als in der Entwicklungsumgebung die Basisklasse geladen wurde.

Tipp

Wenn Sie auf die in der Basisklasse bereits hinzugefügten Steuerelemente noch zugreifen wollen, dann setzen Sie deren Modifier auf `Protected`. Andernfalls sind sie nicht mehr sichtbar, falls die Klassen in unterschiedlichen Assemblys liegen.

7.3 Konfiguration

Konfiguration ist ein wesentlicher Bestandteil des .NET Framework. Sehr viele Einstellungsmöglichkeiten werden in Konfigurationsdateien ausgelagert und können zur Laufzeit eingelesen und ausgewertet werden.

7.3.1 Konfigurationsdateien

Dabei gibt es zwei elementare Konfigurationsdateien:

- Machine.config

  Die *machine.config* ist die zentrale Konfigurationsdatei, die für alle Applikationen auf dem Rechner gültig ist. Die *machine.config* liegt im Verzeichnis *[Windows]\Microsoft.NET\Framework\v4.0.30319\Config*.

- App.config

  Die *app.config* ist die applikationsspezifische Konfigurationsdatei (bei Webanwendungen heißt diese *web.config*). Sie liegt im selben Verzeichnis wie die ausführbare Exe-Datei. Sie kann Einträge aus der *machine.config* überschreiben oder auch erweitern. Während der Entwicklung arbeiten Sie mit der *app.config*. Beim Kompilieren wird die *app.config* ins Ausgabeverzeichnis kopiert und umbenannt in *[Name der Assembly].exe.config*.

Sie sollten Ihre Konfiguration somit in der *app.config* vornehmen. Seien Sie sich bewusst, dass Änderungen an der *machine.config* Auswirkungen auf alle .NET-Anwendungen auf diesem System haben.

Um die Einträge auszulesen, steht Ihnen die Klasse `ConfigurationManager` aus der Bibliothek *System.Configuration.dll* zur Verfügung.

Das Zusammenspiel mit der Konfigurationsdatei soll hier an einem kleinen Beispiel demonstriert werden. In diesem möchte ich die zu ladende Kulturinformation für die Lokalisierung aus der Konfigurationsdatei auslesen.

Dazu muss ich erst einmal eine applikationsspezifische Konfigurationsdatei zum Projekt hinzufügen.

Wenn Sie PROJEKT – NEUES ELEMENT HINZUFÜGEN... wählen, erscheint der Dialog aus Abbildung 7.18. Hier wählen Sie den Eintrag *Anwendungskonfigurationsdatei*.

Achtung

Auch wenn die Programmiersprache Visual Basic 10 nicht zwischen Groß- und Kleinschreibung unterscheidet, sind Konfigurationsdateien sehr wohl case-sensitiv. Konfigurationsdateien sind ja XML-Dateien und XML-Elemente werden grundsätzlich in der CamelCase-Schreibweise definiert.

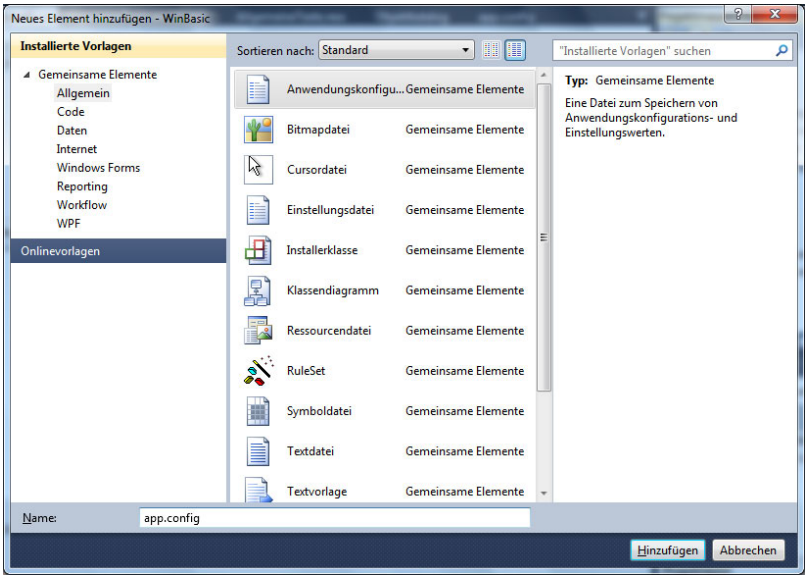


Abbildung 7.18
Anwendungskonfigurationsdatei hinzufügen

Dabei ist das <configuration>-Tag das Root-Element der XML-Datei. Es gibt auch einen Abschnitt <appSettings>, unter dem Sie eigene Einträge hinzufügen können. Ein weiterer sehr häufig verwendeter Abschnitt ist <connectionStrings> zur Speicherung von Datenbankverbindungszeichenfolgen (mehr dazu in Kapitel 9).

Im folgenden Listing 7.8 sehen Sie die Konfigurationsdatei der Windows-Anwendung. Dabei habe ich unter <appSettings> einen neuen Eintrag Kultur hinzugefügt und als Wert en-US.

Listing 7.8 app.config mit Kulturinformation

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Kultur" value="en-US"/>
  </appSettings>
</configuration>
```

Jetzt wollen wir diese Information auslesen und die Kultur des Threads anpassen.

Sie erinnern sich an unseren Code im Konstruktor zum Setzen der Kultur aus Listing 7.7?

Anstatt den String en-US hartkodiert im Konstruktor stehen zu lassen, lesen wir ihn nun aus der *app.config* aus. Dazu müssen wir zuerst einen Verweis auf die Assembly System.Configuration hinzufügen. Danach können wir mit der ConfigurationManager-Klasse den Wert wie folgt auslesen.

```
Dim kultur As String = Configuration. _
    ConfigurationManager.AppSettings("Kultur")
```

Sie müssen nur darauf achten, dass der String, den Sie der Auflistung AppSettings übergeben, wirklich genauso geschrieben wurde wie der key-Eintrag in der Konfigurationsdatei.

Was ist aber, wenn die Kulturinformation geändert werden soll? Besteht auch die Möglichkeit, die Werte zur Laufzeit zu ändern? Dazu werde ich dem Formular ein Menü hinzufügen.

In Listing 7.9 sehen Sie den benötigten Programmcode zum Ändern des Eintrags in der Konfigurationsdatei.

Listing 7.9 Geänderte Kultur in die Konfigurationsdatei schreiben

```
Private Sub SpracheÄndernToolStripMenuItem_Click_
  (ByVal sender As System.Object, ByVal e As EventArgs) _
   Handles SpracheÄndernToolStripMenuItem.Click

   'neue Kultur angeben
   Dim neueKultur As String = InputBox("Neue Kultur")

   Try
     'Thread umstellen
     System.Threading.Thread.CurrentThread.CurrentCulture = _
       New Globalization.CultureInfo(neueKultur)

     System.Threading.Thread.CurrentThread.CurrentUICulture = _
       New Globalization.CultureInfo(neueKultur)
```

```
      'Formular neu aufbauen
      Me.Controls.Clear()
      InitializeComponent()
      'Uhrzeit in der neuen Kultur setzen
      ToolStripStatusLabel1.Text = Now.ToLongDateString
      'Zugriff auf Konfigurationsdatei
      Dim config As Configuration.Configuration = _
        Configuration.ConfigurationManager. _
        OpenExeConfiguration( _
        Configuration.ConfigurationUserLevel.None)
      'Zugriff auf Abschnitt appSettings
      Dim section As Configuration.AppSettingsSection = _
          config.AppSettings
      'neuen Wert setzen
      section.Settings("Kultur").Value = neueKultur
      'Konfiguartion speichern
      config.Save()
    Catch ex As Exception
      MessageBox.Show(ex.Message)
    End Try
End Sub
```

Listing 7.9 (Forts.) Geänderte Kultur in die Konfigurationsdatei schreiben

Hier lese ich zuerst die neue Kulturinformation ein. Danach beginne ich einen Block für die Fehlerbehandlung, damit der Vorgang abgebrochen wird, falls eine nicht gültige Kultur eingegeben wurde.

Anschließend werden die Kulturinformationen des aktuellen Threads umgestellt und das Formular neu initialisiert. Dazu muss die Steuerelementauflistung des Formulars gelöscht werden.

Danach greife ich auf die Konfigurationsdatei zu. Die Klasse `ConfigurationManager` bietet dazu eine statische Methode `OpenExeConfiguration()`, um den Zugriff auf die *app.config* zu erhalten. Über das Objekt, das mir diese Methode zurückliefert, kann ich mir einen Zugriff auf den gewünschten Abschnitt mittels der Eigenschaft `AppSettings` verschaffen. Jetzt muss nur noch der Wert über das Auflistungsobjekt `Settings` neu gesetzt werden und die Konfigurationsdatei mit der Methode `Save()` gespeichert werden.

Achtung

Wenn Sie diese Funktionalität testen, werden Sie feststellen, dass zwar das Layout an die neue Sprache angepasst wird, jedoch keine Änderung in der Konfigurationsdatei durchgeführt wird. Aus der Entwicklungsumgebung heraus funktioniert das auch nicht. Überzeugen Sie sich einfach, indem Sie die ausführbare Exe direkt aufrufen.

7.3.2 Benutzerdefinierte Einstellungen

Das gerade gezeigte Beispiel speichert Einstellungen unabhängig vom Benutzer. Jedoch gibt es auch eine sehr einfache Möglichkeit, benutzerdefinierte Einstellungen zu verwalten.

Mit einem Doppelklick auf MyProject im Projektmappenexplorer kommen Sie zu den Einstellungsmöglichkeiten für das Projekt. Wählen Sie dabei das Register Einstellungen aus.

Jede Einstellung (Setting) wird im .NET Framework 4.0 über den Namen, Datentyp, Anwendungsbereich (auch **Scope** genannt) und Wert gekennzeichnet. Der Anwendungsbereich kann in zweierlei Hinsicht definiert sein. Zum einen kann dieser den Wert *User*, zum anderen den Wert *Application* enthalten. Bei dem Wert *Application* steht die Einstellung allen Benutzern auf dem Rechner zur Verfügung und wird in der Konfigurationsdatei gespeichert. Der Wert *User* bildet eine Abgrenzung zu verschiedenen Benutzern und wird in der Datei *user.config* im Anwendungsdatenverzeichnis gespeichert, zum Beispiel:

```
C:\Dokumente und Einstellungen\user\Lokale Einstellungen\
Anwendungsdaten\Konfiguration\ Konfiguration.vshost.exe_Url_
igbd5ahxfilfyme3nsfxjlp15oehgtyk\1.0.0.0
```

Der Standardwert wird jedoch auch in die Anwendungskonfigurationsdatei eingetragen.

Im folgenden Beispiel werden Sie einen näheren Einblick in die Konfiguration Ihrer Anwendung bekommen. Im Einstellungsregister Ihrer Applikation können Sie dynamische Informationen über Ihre Applikation hinzufügen und ändern. Hierbei stehen Ihnen die Einstellungsmöglichkeiten aus Tabelle 7.11 zur Verfügung:

Tabelle 7.11 Funktionen des Einstellungsregisters

Einstellungsname	Funktion
Name	Der Name der Einstellung, über den Sie später auf die Einstellung zugreifen können
Typ	Hier kann bereits im Vorfeld der Typ der Einstellung festgelegt werden. Dementsprechend ändert sich auch die Einschränkung für einzugebende Werte unter dem Punkt »Wert«.
Bereich	Wie beschrieben, kann hier festgelegt werden, ob diese Information anwenderbezogen ist.
Wert	Hier kann der Wert dieser Einstellung festgelegt werden. Bei Farbeinstellungen kann dies die direkte Auswahl der Farbe in einer Combo-Box darstellen.

Mit der Funktion Synchronisieren können Sie die Werte in die XML-Dateien schreiben oder umgekehrt lesen. Bei der Kompilierung werden die neuen Werte automatisch gespeichert. So wird sichergestellt, dass Ihnen immer die aktuellen Konfigurationsdaten in Ihrer Anwendung zur Verfügung stehen.

Im Folgenden will ich eine benutzerdefinierte Einstellung eintragen, damit der Benutzer die Hintergrundfarbe des Formulars auswählen kann. In Abbildung 7.19 sehen Sie den Eintrag.

Wenn Sie jetzt die Datei *app.config* über den Projektmappenexplorer öffnen, werden Sie feststellen, dass sich ihr Inhalt geändert hat. Dieser stellt sich nun wie in Listing 7.10 dar.

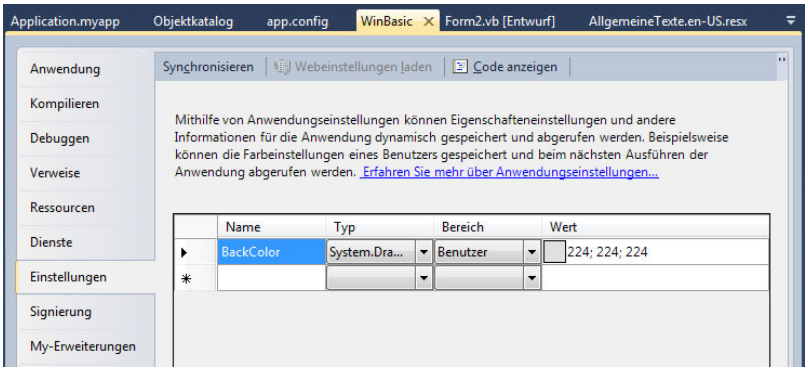


Abbildung 7.19
Einstellung für Hintergrundfarbe

Listing 7.10
app.config mit einem Standardeintrag für die Hintergrundfarbe

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="userSettings"
      type="System.Configuration.UserSettingsGroup,
          System, Version=4.0.0.0,
          Culture=neutral,
          PublicKeyToken=b77a5c561934e089" >
      <section name="WinBasic.My.MySettings"
      type="System.Configuration.ClientSettingsSection,
      System, Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089"
      allowExeDefinition="MachineToLocalUser"
      requirePermission="false" />
    </sectionGroup>
  </configSections>
  <appSettings>
    <add key="Kultur" value="en-US"/>
  </appSettings>
  <userSettings>
    <WinBasic.My.MySettings>
      <setting name="BackColor" serializeAs="String">
        <value>224, 224, 224</value>
      </setting>
    </WinBasic.My.MySettings>
  </userSettings>
  <startup>
    <supportedRuntime version="v4.0"
                sku=".NETFramework, Version=v4.0/>
  </startup>
</configuration>
```

Die Einstellungen wurden nun erfolgreich festgelegt, doch wie können Sie auf diese Einstellungen in Ihrer Anwendung zugreifen?

Dazu bietet Ihnen Visual Basic 2010 die Möglichkeit, die Einstellungen über den `My`-Namensraum zu ermitteln bzw. festzulegen:

```
Me.BackColor = My.Settings.BackColor
```

In unserem kleinen Beispiel will ich noch über ein Menü die Möglichkeit für den Benutzer bieten, die Hintergrundfarbe mit einem Farbauswahldialog zu ändern. Listing 7.11 zeigt den entsprechenden Code.

Listing 7.11 Änderung der Benutzereinstellung

```
Private Sub HintergrundfarbeToolStripMenuItem_Click _
  (ByVal sender As System.Object, ByVal e As EventArgs) _
  Handles HintergrundfarbeToolStripMenuItem.Click

  Dim dialog As New ColorDialog
  dialog.Color = My.Settings.BackColor
  If dialog.ShowDialog = Windows.Forms.DialogResult.OK Then
    Me.BackColor = dialog.Color
    My.Settings.BackColor = dialog.Color
    My.Settings.Save()
  End If
End Sub
```

7.4 Eigene Steuerelemente erstellen

Das Erstellen eigener Steuerelemente (im Englischen *Controls*) wird immer dann erforderlich, wenn die vorhandenen von Visual Studio 2010 nicht mehr für die eigenen Zwecke genügen. Zwar bietet Visual Studio eine Vielzahl von Steuerelementen und in der Regel werden Sie damit auch auskommen. Hier und da kann es jedoch vorkommen, dass man ein Steuerelement mit bestimmten Eigenschaften benötigt, das nicht standardmäßig mit Visual Studio ausgeliefert wird. In diesem Fall können Sie das Steuerelement selbst entwickeln und in eine eigene Bibliothek kompilieren, um es der Toolbox zur Verfügung zu stellen. Man unterscheidet grundsätzlich zwischen drei Arten von selbst entwickelten Controls:

- Das abgeleitete Steuerelement: Bei einem abgeleiteten Steuerelement handelt es sich um ein bereits existierendes Steuerelement, das durch neue Funktionen erweitert wird. Es kann sich z.B. um eine TextBox handeln, bei der Sie nur Zahlen eintragen können.
- Das zusammengesetzte Steuerelement: Hierbei handelt es sich um ein Steuerelement, das aus mehreren einzelnen Controls besteht. Dieses Steuerelement designen Sie entweder direkt in Visual Studio oder Sie leiten es von einer Steuerelementklasse ab. Ein Beispiel für so ein Steuerelement wäre eine Textbox, die gleichzeitig mit einem Label auf dem Formular angelegt wird.
- Komplett selbst erstelltes Steuerelement: Diese Controls werden weder abgeleitet noch gibt es für sie vorgefertigte Lösungen. Diese Controls müssen Sie von Anfang an selbst mit der IDE und GDI+ erstellen. Der Aufwand hierfür ist im Allgemeinen beträchtlich. Daher sollte man sich im Vorfeld umschauen, ob nicht ein Dritthersteller ein entsprechendes Control bereits anbietet.

7.4.1 Abgeleitetes Steuerelement

Das abgeleitete Steuerelement ist wohl das einfachste selbst zu erstellende Steuerelement überhaupt. Hierbei wird das neue Steuerelement anhand eines bereits vorhandenen Steuerelements implementiert und erweitert. Als Ergebnis stehen Ihnen dann alle auch vorher bekannten Funktionen des Steuerelements und Ihre eigenen zur Verfügung.

Als Beispiel nehmen wir ein `Button`-Steuerelement. Das Ergebnis soll, neben den normalen Funktionen, eine neue Funktion haben, die zählt, wie oft der Button während der Laufzeit gedrückt wurde.

Um ein neues Steuerelementprojekt zu starten, wählen Sie Datei/Neu/Projekt und daraufhin unter Visual Basic den Punkt Klassenbibliothek. Benennen Sie das neue Projekt ButtonCount. Nach einem Klick auf den Button OK wird die Projektmappe erstellt.

Um mit der Arbeit beginnen zu können, müssen Sie das Projekt noch mit den benötigten Verweisen ergänzen. Hierzu klicken Sie bitte im Projekt selbst mit Rechtsklick auf den Verweise-Zweig. Suchen Sie im Register .NET die Assembly *System.Windows.Forms.dll* und wählen Sie den Eintrag aus. Neben der Forms Assembly benötigen Sie noch die *System.Drawing.dll*. Nachdem Sie beide Assemblies mit OK hinzugefügt haben, müssen Sie den Namespace festlegen, unter dem das Steuerelement später einmal aufrufbar sein soll. Das ist für die spätere Benutzung von großer Bedeutung, damit Sie oder jemand anderes gezielt nach Ihrem Steuerelement suchen können.

Um die nötigen Einstellungen vorzunehmen, müssen Sie per Rechtsklick im Projekt den Eigenschaften-Dialog aufrufen. Gleich auf der ersten Eigenschaftsseite sehen Sie die Felder Assemblyname und Stamm-Namespace. Schließen Sie die Eigenschaftsseite, nachdem Sie den Namespace angegeben haben – die Einstellungen werden automatisch gespeichert.

Info

Der Stamm-Namespace wird häufig nach dem Unternehmen benannt, in dem das Steuerelement entwickelt wurde. Sie können gegebenenfalls Ihren eigenen Namen oder den Namen Ihres Unternehmens zusammengeschrieben angeben.

Nachdem Sie nun alle Vorbereitungen getroffen haben, kann die eigentliche Programmierung beginnen. Hierzu benennen Sie die Klasse *Class1.vb* zunächst in *ButtonCount.vb* um. Wechseln Sie nun in den Editor von *ButtonCount.vb* und notieren Sie ganz oben in der Klasse folgende Codezeile.

```
Inherits System.Windows.Forms.Button
```

Nun haben Sie Zugriff auf alle bereits vorhandenen Funktionen des `Button`-Objekts. Deklarieren Sie eine Variable mit dem Namen `mcount` als `Integer`-Typ. Diese wird später für die Zählung der Klicks verantwortlich sein.

Um einen Klick auf den Button zu zählen, überschreiben Sie die `OnClick()`-Methode. Die benötigte Zeile lautet dann:

```
Protected Overrides Sub OnClick _
    (ByVal e As System.EventArgs)
```

Nun können Sie mit dem eigentlichen Hochzählen der Variablen beginnen und die Anzahl der Klicks über die Variable `mcount` inkrementieren. Hierzu geben Sie in den Editor folgende Zeilen ein:

```
mcount += 1
MyBase.OnClick(e)
```

Nun fehlt uns nur noch eine Methode, um die Variable `mcount` auslesen zu können. Hierfür können Sie eine schreibgeschützte Eigenschaft mit dem Namen `Count` deklarieren.

Den vollständigen Code für das Projekt sehen Sie in Listing 7.12.

Listing 7.12 Code des Count-Buttons

```
Public Class ButtonCount
  Inherits System.Windows.Forms.Button
  Private mcount As Integer = 0

  Protected Overrides Sub OnClick (ByVal e As System.EventArgs)
    mcount += 1
    MyBase.OnClick(e)
  End Sub

  Public ReadOnly Property Count() As Integer
    Get
      Return mcount
    End Get
  End Property
End Class
```

Das erstellte Steuerelement sieht in der Toolbox standardmäßig aus wie ein Zahnrad. Falls Sie mehrere eigene Steuerelemente erstellen, würden sich deren Icons in der Toolbox nicht voneinander unterscheiden. Natürlich können Sie auch für dieses Problem Abhilfe schaffen.

Wechseln Sie dazu in die Projekteigenschaften und wählen Sie den Registereintrag Ressourcen. Hier können Sie über den Menüeintrag Ressource hinzufügen das gewünschte Bild auswählen. Sie können es im Nachhinein noch auf die Standard-Icon-Größe von 16 * 16 Pixel bringen. Nachdem das Icon eingefügt und angepasst wurde, müssen Sie es im Eigenschaftenfenster auswählen und den Eintrag Buildvorgang auf eingebettete Ressource umstellen. Beim nächsten Kompilieren steht das Icon dann in der Bibliothek zur Verfügung. Um es noch als Toolbox-Icon zu kennzeichnen, müssen Sie folgendes Attribut für die Klasse ButtonCount setzen:

```
<System.Drawing.ToolboxBitmap(GetType(ButtonCount), "icon.jpg")>
```

Dieses Attribut müssen Sie vor die eigentliche Deklaration schreiben. `ButtonCount` steht in diesem Fall für die Klasse und *icon.jpg* für den Namen des Icons.

Nachdem nun der Code fertiggestellt wurde, können Sie das Projekt kompilieren. Die erzeugte Bibliothek können Sie nun in die Toolbox von Visual Studio einbinden und so den neuen Button Ihren eigenen Projekten zur Verfügung stellen. Hierfür starten Sie eine neue Windows-Anwendung, klicken mit der rechten Maustaste auf die Toolbox und wählen unter Elemente auswählen die Assembly aus dem ButtonCount-Ordner. Danach sollten Sie in der Toolbox ein neues Steuerelement mit dem Namen ButtonCount und dem ausgewählten Icon sehen. Dieses können Sie nun, wie gewohnt, auf Ihr Formular ziehen.

Im ersten Moment scheint es so, als habe sich nichts am eigentlichen Button verändert. Doch beim genaueren Hinsehen können Sie im Eigenschaftsfenster die Eigenschaft Count mit dem Wert 0 finden. Diese Eigenschaft können Sie natürlich nicht verändern, deshalb erscheint sie grau. Wenn Sie das Projekt nun starten, wird der Zähler bei jedem Klick um eins hochgezählt. Das Ergebnis können Sie zum Beispiel ganz einfach in einer `MessageBox` anzeigen lassen:

```
MessageBox.Show(Me.ButtonCount1.Count)
```

Der Rückgabewert der `MessageBox` zeigt zum jeweiligen Zeitpunkt an, wie oft der Button während der Programmausführung gedrückt wurde.

7.5 MDI-Formulare

Bei der Programmierung von Windows-Anwendungen gibt es folgende gebräuchliche Layoutoptionen:

- SDI – Single Document Interface
- MDI – Multiple Document Interface
- Explorerstil

Während bei einer SDI-Anwendung nur eine einzige Instanz eines Formulars verfügbar ist, können bei MDI mehrere Instanzen gleichzeitig angezeigt werden.

Anwendungen im Stile des Windows Explorers teilen in der Regel das Formular in zwei Bereiche. Im (meistens) linken Bereich gibt es eine Auswahlmöglichkeit, sehr oft mittels eines `TreeView`-Steuerelements, während im rechten Bereich die entsprechenden Daten angezeigt werden.

MDI-Anwendungen bestehen dabei aus einem MDI-Container oder auch dem MDI-Parent. Pro Anwendung kann es maximal einen einzigen MDI-Parent geben. Alle MDI-Childs, die untergeordneten Fenster, bewegen sich im Rahmen dieses MDI-Containers.

MDI-Anwendungen bieten die Möglichkeit, mehrere MDI-Childs innerhalb des MDI-Containers anzuzeigen und zu laden. Dabei können die MDI-Childs auch unterschiedliche Instanzen von Formularen sein. Sollte ein MDI-Child ein eigenes Menü besitzen, integriert es sich standardmäßig in das Menü des übergeordneten MDI-Containers. Sie können jedoch über die Eigenschaft `MergeAction` eines Hauptmenüpunkts definieren, wo das Menü tatsächlich angezeigt wird.

Der MDI-Container selbst bietet zumeist ein Fenstermenü, über das die Kindfenster gesteuert werden können und Layouteinstellungen der Kindfenster gesetzt werden.

7.5.1 Erstellen einer MDI-Beispielanwendung

Jetzt wollen wir noch in einem kleinen Beispiel eine MDI-Anwendung erstellen.

Dazu legen wir ein neues Projekt vom Typ Windows Forms-Anwendung an und benennen dieses Projekt *MDIBeispiel*.

Die standardmäßig angelegte *Form1* wollen wir umbenennen in *MDIContainer* und durch das Setzen der Eigenschaft `IsMdiContainer` auf `True` auch als solchen definieren.

Außer den Elementen aus dem Register *Menüs & Symbolleisten* finden Sie in der Regel keine weiteren Steuerelemente auf einem MDI-Container. In unserem Beispiel will ich eine `MenuStrip` zum Formular hinzufügen. Das Menü besteht dabei aus einem Hauptmenüpunkt *Datei* und zwei Unterpunkten *Neu* und *Beenden*.

Mittels *Neu* können neue MDI-Childs geladen werden, während *Beenden* sinngemäß die Anwendung beendet.

Als Zweites füge ich dem Projekt ein neues Formular hinzu und vergebe den Namen *frmChild*. Diesem Formular füge ich eine *TextBox* hinzu, docke sie im Formular an und setze die Eigenschaft *MultiLine* auf *True*. Den Namen der `TextBox` setze ich auf *txtInhalt*.

Das folgende Listing 7.13 zeigt den Programmcode unseres MDI-Containers.

Listing 7.13
Code des MDI-Containers

```
Public Class MDIContainer

  Dim zaehler As Integer = 1
  Private Sub BeendenToolStripMenuItem_Click _
    (ByVal sender As System.Object, ByVal e As EventArgs) _
     Handles BeendenToolStripMenuItem.Click

    Me.Close()
  End Sub

  Private Sub NeuToolStripMenuItem_Click _
    (ByVal sender As System.Object, ByVal e As EventArgs) _
     Handles NeuToolStripMenuItem.Click

    Dim dialog As New frmChild()
    dialog.MdiParent = Me
    dialog.Text = "Dokument " & zaehler.ToString()
    zaehler += 1
    dialog.Show()

  End Sub
End Class
```

Sie sehen, dass zum Neuanlegen eines Kindfensters das Formular ganz normal instanziiert wird. Bevor das Formular jedoch angezeigt wird, wird die Eigenschaft `MdiParent` auf den aufrufenden Container gesetzt. Danach wird nur noch ein Text für die Fensterüberschrift des neuen Formulars gesetzt.

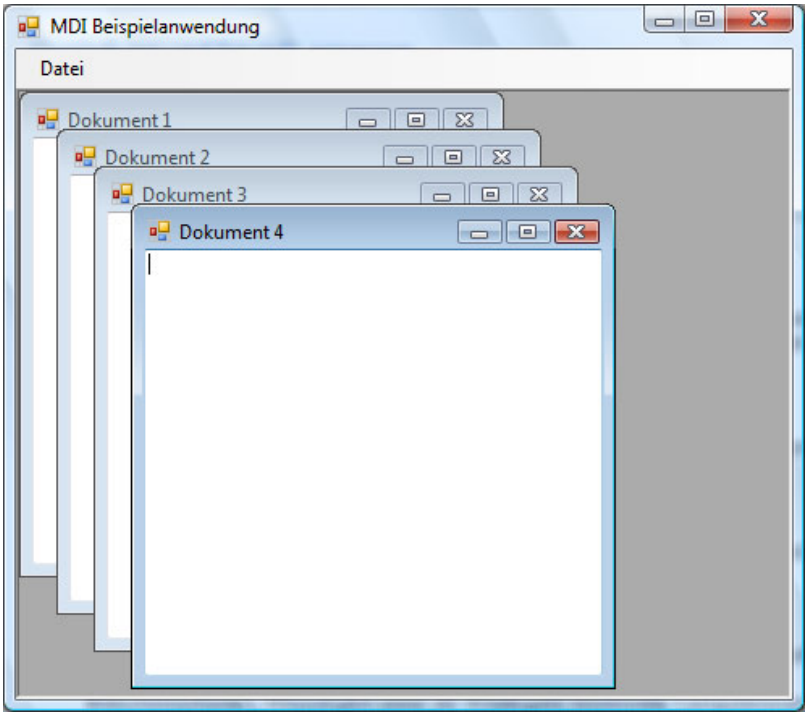


Abbildung 7.20
MDI-Anwendung

Um nun noch Zugriff auf den Inhalt der `TextBox` im aktiven Fenster zu bekommen, benötigen Sie folgende Programmzeile:

```
Dim inhalt As String = CType(Me.ActiveMdiChild, _
          frmChild).txtInhalt.Text
```

Die Eigenschaft `ActiveMdiChild` bietet dabei Zugriff auf das aktuelle Kindfenster.

So, nun fehlt uns nur noch das Fenstermenü. Fügen Sie dem Menü des MDI-Containers noch einen Hauptmenüpunkt *Fenster* hinzu.

Dann setzen Sie die Eigenschaft `MdiWindowListItem` des `MenuStrip` mittels der `DropDownList` auf den Namen des gerade angelegten Fenstermenüs.

Das Fenstermenü sehen Sie in Abbildung 7.21.

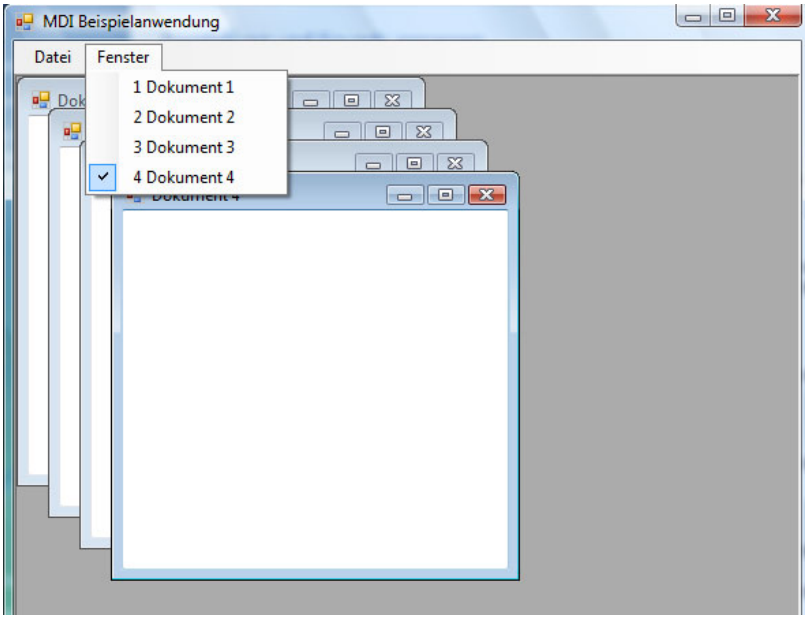


Abbildung 7.21 MDI-Anwendung mit Fenstermenü

Sehr häufig findet man in Fenstermenüs auch noch Möglichkeiten, um die Fensteranordnung zu definieren.

Gewöhnlich gibt es dazu die Optionen:

- Untereinander
- Nebeneinander
- Überlappend
- Symbole anordnen

Diese vier Menüpunkte füge ich noch zum Fenstermenü hinzu und setze über die Methode `LayoutMdi()` die entsprechende Option. Da die Option *Symbole anordnen* nur für minimierte Kindfenster gilt, benötigen Sie auch noch Code zum Minimieren der Fenster für diese Option bzw. zum Wiederherstellen bei den anderen Optionen.

Listing 7.14 zeigt den entsprechenden Code.

Listing 7.14 Code zur Einstellung des Fensterlayouts

```
Private Sub
  UntereinandernToolStripMenuItem_Click _
  (ByVal sender As System.Object, ByVal e As EventArgs) _
   Handles UntereinandernToolStripMenuItem.Click

  For Each f As Form In MdiChildren
    f.WindowState = FormWindowState.Normal
  Next
  LayoutMdi(MdiLayout.TileHorizontal)
End Sub
```

Listing 7.14 (Forts.)
Code zur Einstellung des Fensterlayouts

```
Private Sub
  NebeneinanderToolStripMenuItem_Click _
  (ByVal sender As System.Object, ByVal e As EventArgs) _
   Handles NebeneinandernToolStripMenuItem.Click

  For Each f As Form In MdiChildren
    f.WindowState = FormWindowState.Normal
  Next
  LayoutMdi(MdiLayout.TileVertical)
End Sub

Private Sub ÜberlappendToolStripMenuItem_Click _
  (ByVal sender As System.Object, ByVal e As EventArgs) _
   Handles ÜberlappendToolStripMenuItem.Click

  For Each f As Form In MdiChildren
    f.WindowState = FormWindowState.Normal
  Next
  LayoutMdi(MdiLayout.Cascade)
End Sub
Private Sub _
  SymboleAnordnenToolStripMenuItem_Click _
  (ByVal sender As System.Object, ByVal e As EventArgs) _
   Handles SymboleAnordnenToolStripMenuItem.Click

  For Each f As Form In MdiChildren
    f.WindowState = FormWindowState. Minimized
  Next
  LayoutMdi(MdiLayout. ArrangeIcons)
End Sub
```

Die fertige Anwendung sehen Sie in Abbildung 7.22.

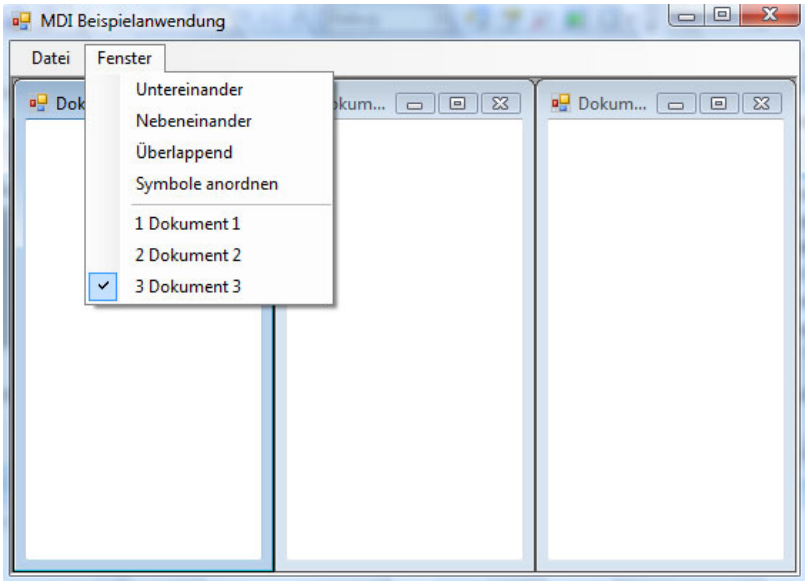


Abbildung 7.22
Fertige MDI-Anwendung

7.5.2 Clipboard

Vor allem bei MDI-Anwendungen benötigen Sie auch sehr oft Zugriff auf die Zwischenablage. Dafür steht im .NET Framework das `Clipboard`-Objekt zur Verfügung. Diese seit der ersten .NET Version erweiterte Klasse kann nun auch ermitteln, um was für ein Format es sich bei den in der Zwischenablage befindlichen Daten handelt.

Das folgende Beispiel in Listing 7.15 verdeutlicht, wie sich die Klasse `Clipboard` zwischen reinem Text und Bildern unterscheidet.

Die Anwendung prüft zunächst, welches Format die Daten in der Zwischenablage besitzen. Die Beispielanwendung verwendet ASCII-Text und eine Bilddatei. Je nachdem, ob es sich um den Text oder um das Bild handelt, wird der Inhalt der Zwischenablage entweder in eine `ListBox` eingefügt oder in eine `PictureBox`.

Listing 7.15 Verwenden der Zwischenablage

```
Private Sub Button1_Click (ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles Button1.Click

  If Clipboard.ContainsImage Then
    PictureBox1.Image = Clipboard.GetImage
  Else
    If ContainsText Then
      lstClipboard.Items.Add(Clipboard.GetText())
    End If
  End If
End Sub
```

Ich denke, der Programmcode ist selbst erklärend. Zum Hinzufügen in die Zwischenablage stehen entsprechende Set-Methoden, wie zum Beispiel `SetText()`, zur Verfügung. Um die Zwischenablage zu löschen, können Sie die Methode `Clear()` verwenden.

7.6 Drag&Drop

Visual Basic 10 unterstützt auch *Drag&Drop* für alle Steuerelemente. Die Methode `DoDragDrop()` ermöglicht es Ihnen, Daten zwischen verschiedenen Steuerelementen zu verschieben.

In diesem Beispiel sehen Sie, wie Inhalt zwischen zwei Textboxen mit Drag&Drop ausgetauscht werden kann.

In Listing 7.16 wird der Drag&Drop-Vorgang durch die `DoDragDrop()`-Methode der Textbox `txtQuelle` ausgelöst. Wenn der Mauszeiger über ein Steuerelement fährt, wird das `DragEnter`-Ereignis aufgerufen. Hier wird festgelegt, ob die Daten in diesem Steuerelement eingefügt werden dürfen. Erst wenn der Anwender die Maustaste über dem Steuerelement loslässt, wird das `DragDrop`-Ereignis aufgerufen und die Daten werden aus der Quell-Textbox in das neue Steuerelement kopiert.

Damit die Textbox `txtZiel` als Ziel eines Drag&Drop-Vorgangs benutzt werden kann, muss die Eigenschaft `AllowDrop` des Steuerelements auf `True` gesetzt werden.

Listing 7.16 Beispielcode für Drag&Drop

```
Private Sub txtQuelle_MouseDown(ByVal sender As Object, _
   ByVal e As MouseEventArgs) Handles txtQuelle.MouseDown

   txtQuelle.SelectAll()
   txtQuelle.DoDragDrop(txtQuelle.SelectedText, _
     DragDropEffects.Move Or DragDropEffects.Copy)
End Sub

Private Sub txtZiel_DragEnter(ByVal sender As Object, _
   ByVal e As DragEventArgs) Handles txtZiel.DragEnter

    e.Effect = DragDropEffects.Move
End Sub

Private Sub txtZiel_DragDrop(ByVal sender As Object, _
   ByVal e As DragEventArgs) Handles txtZiel.DragDrop

  txtZiel.Text = e.Data.GetData _
    (DataFormats.Text).ToString
End Sub
```

7.7 Asynchrone Verarbeitung

Asynchrone Programmierung bietet die Möglichkeit, dass man nach dem Aufruf einer zeitaufwändigen Methode sofort weiterarbeiten kann, ohne dass der aufrufende Prozess blockiert wird, weil er auf die Abarbeitung der aufgerufenen Methode warten muss. Anstatt den Eindruck eines eingefrorenen Formulars zu erwecken, reagiert das Formular weiter auf Benutzereingaben und die aufgerufene Methode wird im Hintergrund abgearbeitet. Dies nennt man auch Multithreading-Programmierung, da die auszuführenden Operationen quasi parallel in eigenen Threads abgearbeitet werden.

7.7.1 Asynchrones Pattern

Asynchrone Muster können dabei unter .NET unterschiedlich implementiert werden. Die `BackgroundWorker`-Komponente implementiert dieses Muster und wird im zweiten Teil dieses Abschnitts betrachtet.

Wir wollen aber zuerst an einem einfachen Beispiel demonstrieren, wie das asynchrone Muster selbst implementiert werden kann.

Dafür benötigen wir zuerst einen Delegate, welcher der Signatur der Methode entspricht, die asynchron aufgerufen werden soll.

In unserem einfachen Beispiel wollen wir eine Funktion mit einem `Integer` als Rückgabewert und einem `Integer`-Eingabeparameter aufrufen. Der Delegate muss somit wie folgt aussehen:

```
Private Delegate Function AsyncDelegate _
    (ByVal i As Integer) As Integer
```

Danach definieren wir eine Objektvariable von diesem Delegate:

```
Private caller As AsyncDelegate = Nothing
```

Um die Methode asynchron aufzurufen, muss der `AsyncDelegate` instanziiert werden und dabei die Adresse der gewünschten Methode im Konstruktor angegeben werden. Da wir auch den Rückgabewert der Funktion auswerten wollen, benötigen wir auch eine sogenannte Callback (Rückruf)-Methode. Die Rückrufmethode ist als Prozedur zu definieren, welche als Parameter ein Objekt vom Typ `IAsyncResult` erhält.

Zum Schluss rufen wir dann mit der Methode `BeginInvoke()` des Delegate die gewünschte Methode asynchron auf. Als Parameter benötigen wir den `Integer`-Parameter sowie die Adresse der Rückrufmethode.

Listing 7.17 Asynchroner Aufruf einer Methode

```
caller = New AsyncDelegate (AddressOf LongRunningMethod)
Dim callback As New AsyncCallback(AddressOf ShowResult)
caller.BeginInvoke(3, callback, Nothing)
```

In der Callback-Routine (Listing 7.18) kann dann mit der Methode `EndInvoke()` der Rückgabewert ausgewertet werden.

Listing 7.18 Auswerten des Rückgabewerts in der Callback-Routine

```
Sub ShowResult(ByVal ar As IAsyncResult)
  Dim ergebnis As Integer = caller.EndInvoke(ar)
  MessageBox.Show(ergebnis.ToString())
End Sub
```

In Listing 7.19 sehen Sie die komplette Implementierung des asynchronen Pattern:

Listing 7.19 Implementierung des asynchronen Pattern

```
Public Class Form1

  Private Delegate Function AsyncDelegate _
    (ByVal i As Integer) As Integer

  Private caller As AsyncDelegate = Nothing

  Private Sub Form1_Load(ByVal sender As System.Object, _
     ByVal e As System.EventArgs) Handles MyBase.Load

    caller = New AsyncDelegate(AddressOf LongRunningMethod)
    Dim callback As New AsyncCallback(AddressOf ShowResult)
    caller.BeginInvoke(3, callback, Nothing)
  End Sub

  Function LongRunningMethoded(ByVal i As Integer) As Integer
    System.Threading.Thread.Sleep(2000)
    Return i * 2
  End Function

  Sub ShowResult(ByVal ar As IAsyncResult)
    Dim ergebnis As Integer = caller.EndInvoke(ar)
    MessageBox.Show(result.ToString())
  End Sub
End Class
```

7.7.2 BackgroundWorker-Komponente

Die BackgroundWorker-Komponente stellt eine Implementierung des asynchronen Musters dar, bei der man sich um gewisse Systemdetails keine Gedanken machen muss.

Der BackgroundWorker stellt eine Methode RunWorkerAsync() zur Verfügung, um einen asynchronen Aufruf zu starten. Durch den Aufruf dieser Methode wird das DoWork-Event der Komponente ausgelöst. Beim Aufruf von RunWorkerAsync() können Sie einen Parameter vom Typ Object übergeben, falls der Hintergrundvorgang diesen benötigt.

Im DoWork-Ereignis erfolgt dann der tatsächliche Aufruf an die zeitaufwändige Methode LongRunningMethod(). Beachten Sie in diesem Event, dass Sie nicht direkt auf die BackgroundWorker-Komponente zugreifen, sondern über den Übergabeparameter sender den Zugriff auf die Komponente herstellen. Andernfalls bekommen Sie bei mehreren gleichzeitigen Zugriffen Probleme.

Um mit dieser Komponente den Rückgabewert auszuwerten, müssen wir auf das Ereignis RunWorkerCompleted reagieren. Dieses wird nach dem kompletten Abarbeiten des DoWork()-Ereignishandlers ausgelöst. Überprüfen Sie allerdings im RunWorkerCompleted-Ereignis zuerst, ob nicht ein Fehler bei der Verarbeitung aufgetreten ist. Dazu steht Ihnen über den Übergabeparameter die Error-Eigenschaft zur Verfügung.

Innerhalb des RunWorkerCompleted-Ereignishandlers kann das Funktionsergebnis ausgewertet und auf der Oberfläche ausgegeben werden.

Achtung

Wenn Sie Eigenschaften von Steuerelementen setzen wollen, dann tun Sie das ausschließlich im ProgressChanged- und im RunWorkerCompleted-Ereignis. Wenn Sie dies im DoWork-Ereignis tun, bekommen Sie Laufzeitfehler wegen Synchronisationsproblemen.

Die BackgroundWorker-Komponente stellt mittels der Methode CancelAsync() auch Funktionalität zur Verfügung, um den asynchronen Aufruf abzubrechen. Das funktioniert aber nur, wenn die Eigenschaft WorkerSupportsCancellation auf True gesetzt wird.

Listing 7.20 zeigt dabei den asynchronen Aufruf der LongRunningMethod()-Methode mittels der BackgroundWorker-Komponente.

Listing 7.20
BackGroundWorker-Komponente

```
Public Class Form1

  Function LongRunningMethod(ByVal i As Integer) As Integer
    System.Threading.Thread.Sleep(2000)
    Return i * 2
  End Function
  Private Sub BackgroundWorker1_DoWork(ByVal sender As Object,
     ByVal e As System.ComponentModel.DoWorkEventArgs) _
     Handles BackgroundWorker1.DoWork
```

Listing 7.20 (Forts.) BackGroundWorker-Komponente

```
    Dim bw As System.ComponentModel.BackgroundWorker = _
    DirectCast(sender, System.ComponentModel.BackgroundWorker)
    e.Result = LongRunningMethod(Convert.ToInt32(e.Argument))

    If bw.CancellationPending Then
      e.Cancel = True
    End If
  End Sub
  Private Sub
    BackgroundWorker1_RunWorkerCompleted
    (ByVal sender As Object, _
     ByVal e As System.ComponentModel. _
          RunWorkerCompletedEventArgs) _
     Handles BackgroundWorker1.RunWorkerCompleted

    If (e.Error IsNot Nothing) Then
      MessageBox.Show(e.Error.Message)
    Else
      If e.Cancelled = True Then
         MessageBox.Show("Aktion abgebrochen!")
      Else
         MessageBox.Show(e.Result.ToString())
      End If
    End If
  End Sub

  Private Sub btnStart_Click(ByVal sender As Object, _
     ByVal e As System.EventArgs) Handles btnStart.Click
     BackgroundWorker1.RunWorkerAsync(5)
  End Sub
  Private Sub btnCancel_Click (ByVal sender As Object, _
     ByVal e As System.EventArgs) Handles btnCancel.Click
        BackgroundWorker1.CancelAsync()
  End Sub
End Class
```

8

WPF – eine kurze Einführung

Nachdem es in Kapitel 7 gerade um Windows Forms gegangen ist, will ich Ihnen in diesem etwas kleineren Kapitel einen Überblick zu WPF geben.

WPF steht für die Windows Presentation Foundation und wurde mit der Framework-Version 3.0 eingeführt. WPF ist komplett in Visual Studio 2010 integriert und stellt momentan die erste Wahl zur Oberflächengestaltung dar.

8.1 Was ist WPF?

WPF ist eine neue Technologie zur Erstellung grafischer Benutzeroberflächen, 2D- und 3D-Grafiken, Animationen sowie von Dokumenten.

Bei WPF werden vor allem auch Designer in den Entwicklungsprozess mit eingebunden. Tatsächlich gehen die Möglichkeiten, die WPF in punkto Design bietet, weit über die von Windows Forms hinaus. Dabei stehen den Designern Tools zur Verfügung, die komplett von Visual Studio gelöst sind. Mit den Expression-Tools bietet Microsoft leistungsfähige Werkzeuge, um WPF-Oberflächen zu erstellen.

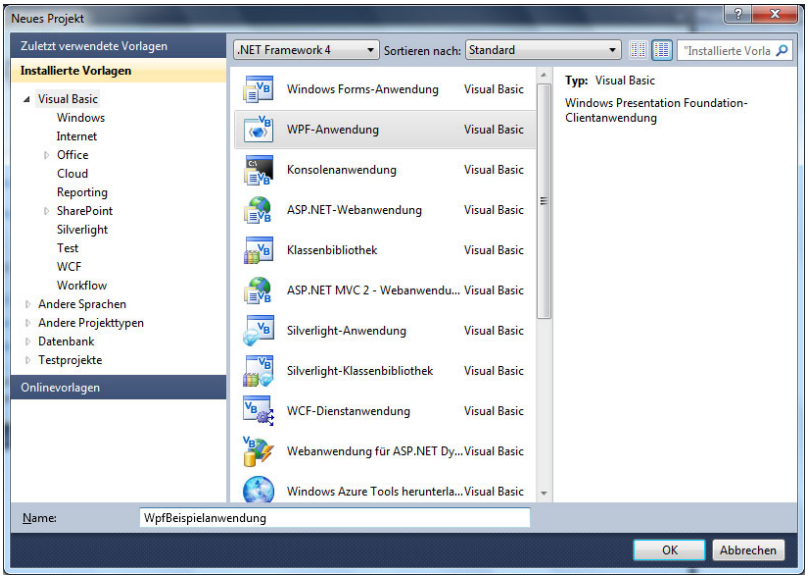
Die Erstellung von Oberflächen geschieht dabei deklarativ mit *XAML*, einer XML-basierten Syntax. XAML steht für eXtensible Application Markup Language. XAML bietet als Markupsprache die Möglichkeit, das komplette Layout aller Elemente eines Window zu definieren und sogar Animationen zu erstellen. Mittels Attributen können sogar Ereignisse und Methoden für WPF-Elemente definiert werden. Für die tatsächliche Implementierung der Logik und der Methoden benötigen Sie dann selbstredend Programmcode in sogenannten Code-Behind-Dateien (den ASP.NET-Entwicklern sollte das bekannt vorkommen).

Für jedes XAML-Element gibt es im .NET Framework eine entsprechende Klasse innerhalb der WPF-Klassenbibliothek.

8.2 WPF-Beispielanwendung

Innerhalb von Visual Studio 2010 steht für WPF-Anwendungen eine eigene Projektvorlage zur Verfügung, wie Sie in Abbildung 8.1 sehen.

Abbildung 8.1
WPF-Projektvorlage



Wenn Sie ein Projekt von diesem Typ anlegen, wird Ihnen folgende Oberfläche, die Sie in Abbildung 8.2 sehen, präsentiert. Sie besteht im Hauptbereich aus zwei Frames, wobei im oberen das Window – unter WPF ist der Ausdruck »Forms« weniger gebräuchlich – angezeigt wird, während im unteren Teil der entsprechende XAML-Code dargestellt wird. Durch Markierung eines Elements im XAML-Code wird das entsprechende Element im oberen Bereich optisch hervorgehoben. Über einen Slider am linken Rand der oberen Ansicht können Sie die Größe der Layoutanzeige skalieren.

Nun ja, mit welchem anderen Einstiegsbeispiel als *HelloWorld* sollte man denn schon beginnen.

Ich beginne, indem ich aus der Toolbox – Sie werden bemerken, dass Sie teilweise andere Steuerelemente vorfinden als bei Windows Forms – einen `Button` auf mein Formular ziehe. Der im unteren Bereich angezeigte XAML-Code passt sich, genauso wie die Designer-Dateien in Windows Forms, automatisch an. Wenn Sie zum Beispiel die Beschriftung des Buttons ändern wollen, können Sie das wie gewohnt auch über das Eigenschaftenfenster tun, das jedoch ein bisschen anders aussieht (siehe Abbildung 8.3). Suchen Sie aber nicht nach einer Eigenschaft `Text`, die entsprechende Eigenschaft heißt hier `Content`.

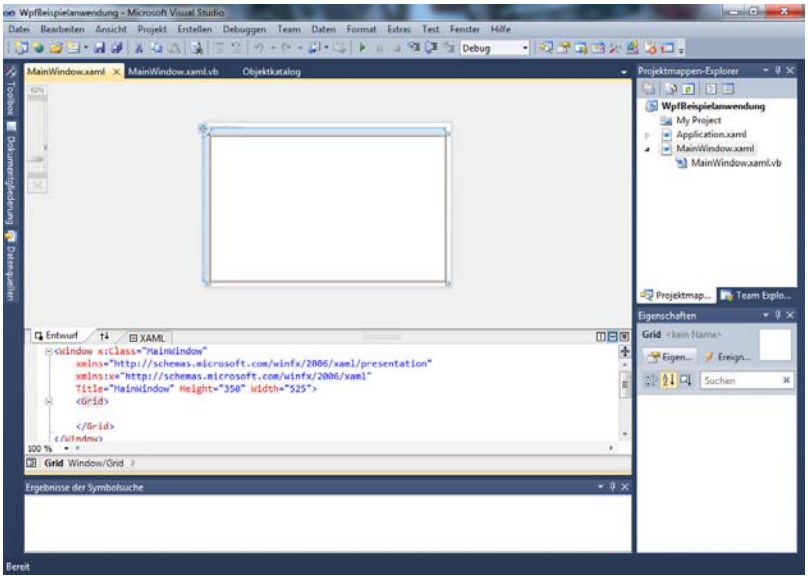


Abbildung 8.2
WPF-Anwendung

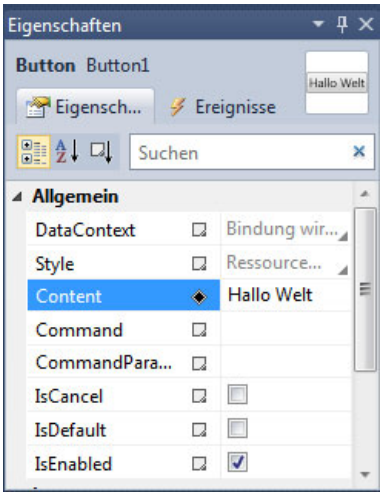


Abbildung 8.3
WPF-Button im Eigenschaftenfenster

Der XAML-Code sieht danach in etwa wie in Listing 8.1 dargestellt aus.

```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presen-
    tation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MainWindow" Height="350" Width="525">
  <Grid>
    <Button Content="Hallo Welt" Height="23"
     HorizontalAlignment="Left" Margin="34,36,0,0"
     Name="Button1" VerticalAlignment="Top" Width="75" />
  </Grid>
</Window>
```

Listing 8.1
XAML-Code eines Window mit Button

Der `Button` wurde dabei innerhalb eines `Grid`-Elements eingebettet.

Sie können jetzt schon Ihre Applikation starten.

Als Nächstes stellt sich die Frage: Wie komme ich zu meinem Ereignishandler, wenn der Benutzer einen Klick auf den `Button` durchführt?

Das funktioniert analog zu Windows Forms. Klicken Sie einfach doppelt auf den `Button` und der Handler wird in der Code-Datei automatisch angelegt.

Listing 8.2 Ereignisroutine für den Buttonklick

```
Class MainWindow

  Private Sub Button1_Click _
    (ByVal sender As System.Object, _
     ByVal e As System.Windows.RoutedEventArgs) _
     Handles Button1.Click
     MessageBox.Show("Hello World")
  End Sub
End Class
```

Wie Sie in Listing 8.2 erkennen können, handelt es sich dabei um den üblichen Programmcode. Alternativ hätten Sie den Ereignishandler auch als Attribut in XAML definieren können:

```
<Button Content="Hallo Welt" Height="23"
     Click="Button1_Click" HorizontalAlignment="Left"
     Margin="34,36,0,0" Name="Button1"
     VerticalAlignment="Top" Width="75" />
```

Durch das Setzen dieses zusätzlichen `Click`-Attributs hätten Sie sich die `Handles`-Anweisung in Listing 8.2 sparen können.

So weit, so gut. Ich hoffe, dass Sie damit einen ersten Eindruck in die neue WPF-Welt bekommen haben.

8.3 Container und Steuerelemente

Die Basis von WPF-Anwendungen sind natürlich wiederum Steuerelemente, die innerhalb von Containern liegen.

8.3.1 Container

Wie Sie im vorigen Beispiel gesehen haben, wurde der Button innerhalb eines `Grid` angelegt. Das bedeutet, dass Steuerelemente immer innerhalb eines Containers liegen.

Außer dem `Grid` gibt es noch andere gebräuchliche Container in WPF, die ich in Tabelle 8.1 kurz vorstellen möchte.

Tabelle 8.1
WPF-Container

Container Element	Zweck
`Grid`	Eignet sich für die relative Positionierung der Elemente innerhalb eines Containers
`DockPanel`	Eignet sich für Steuerelemente, die an den Seiten eines Containers angedockt werden sollen
`StackPanel`	Eignet sich für die horizontale oder vertikale Anordnung von Steuerelementen innerhalb eines Containers
`Canvas`	Eignet sich für die absolute Positionierung der Elemente innerhalb eines Containers

8.3.2 Steuerelemente

Viele Steuerelemente unter WPF sind vergleichbar mit herkömmlichen Windows Form-Steuerelementen, jedoch gibt es auch einige zusätzliche Steuerelemente.

Sämtliche WPF-Steuerelemente liegen im Namensraum `System.Windows.Controls` und erben von einer Basisklasse `UIElement`.

Zur Demonstration will ich unserer Beispielanwendung noch ein Menü hinzufügen.

Ziehen Sie dazu aus der ToolBox ein Menü auf das Formular. Über die Eigenschaft `Items` können Sie dann die Hauptmenüeinträge erfassen. Die Untermenüeinträge können jeweils über die Eigenschaft `Items` des Hauptmenüs erfasst werden.

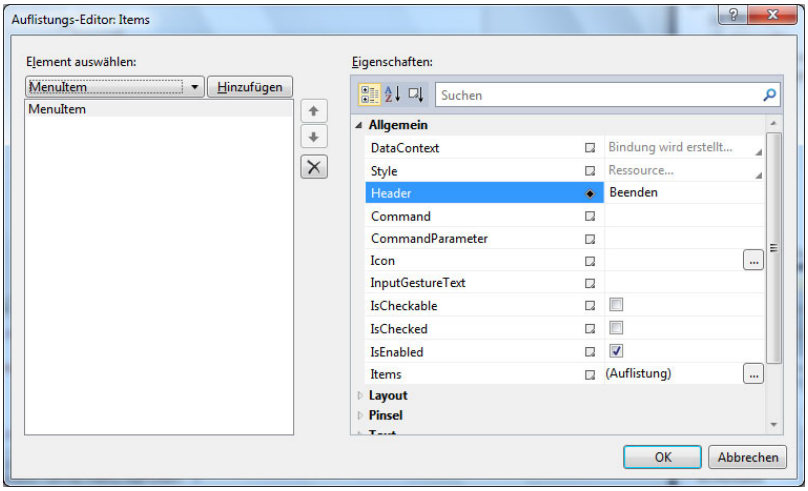


Abbildung 8.4
Menü-Einträge definieren

Der zugehörige XAML-Code ist in Listing 8.3 dargestellt.

Listing 8.3 Window mit Menü

```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presen-
    tation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    Title="MainWindow" Height="350" Width="525">
    <Grid>
     <Button Content="Hallo Welt" Height="23"
              HorizontalAlignment="Left"
              Margin="34,36,0,0"
              Name="Button1"
              VerticalAlignment="Top"
              Width="75" />
        <Menu Height="23"
              HorizontalAlignment="Left"
              Margin="0,0,0,0" Name="Menu1"
              VerticalAlignment="Top"
              Width="70">
            <MenuItem Header="Datei">
                <MenuItem Header="Beenden"/>
            </MenuItem>
        </Menu>
    </Grid>
</Window>
```

8.4 Animationen

Zum Abschluss möchte ich noch mittels einer kleinen Animation die Mächtigkeit von WPF demonstrieren.

Animationen können dabei sehr einfach durch *Storyboards* definiert werden. Innerhalb von Visual Studio 2010 werden Sie leider nicht visuell bei der Erstellung eines Storyboards unterstützt. Das ist den Expression-Produkten vorbehalten. Mit einigen Zeilen XAML können wir aber ein kleines Storyboard selbst erstellen, wie Sie in Listing 8.4 sehen. Dabei soll die Schriftgröße des `Button` innerhalb von drei Sekunden von 14 auf 30 Punkte geändert werden.

Listing 8.4 Window mit Animation

```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presen-
    tation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MainWindow" Height="400" Width="400">
  <Canvas>
    <Button Margin="106.25,112.5,96.25,126.25"
            Name="Button1">
            Hallo Welt
      <Button.Triggers>
       <EventTrigger RoutedEvent="Window.Loaded">
```

Listing 8.4 (Forts.)
Window mit Animation

```
        <BeginStoryboard>
         <Storyboard>
          <DoubleAnimation
           Storyboard.TargetName="Button1"
           Storyboard.TargetProperty="(Button.FontSize)"
           From="14" To="30" Duration="0:0:3">
          </DoubleAnimation>
         </Storyboard>
        </BeginStoryboard>
       </EventTrigger>
      </Button.Triggers>
     </Button>

     <Menu Name="Menu1" VerticalAlignment="Top">
      <MenuItem Header="Datei">
       <MenuItem Header="Beenden"
                 Click="MenuItem_Click"/>
       </MenuItem>
      </Menu>
    </Canvas>
</Window>
```

In Listing 8.4 sehen Sie, dass innerhalb des `Button` ein Element `<Button.Trigger>` definiert ist. Prinzipiell können Sie, wie bereits gezeigt, Ihre Ereignishandler auch im Programmcode definieren. Trigger bieten jedoch eine zusätzliche Möglichkeit. Sie können über Trigger nämlich beim Auftreten eines bestimmten Ereignisses die Eigenschaften von Elementen verändern – was wir hier ausnutzen wollen.

Das Ereignis, auf das wir reagieren wollen, wird über das Attribut `RoutedEvent` des `<EventTrigger>`-Elements definiert. In diesem Fall handelt es sich um das Laden des Window.

Danach erfolgt bereits die Definition des Storyboard. Dabei wird eine sogenannte `DoubleAnimation` durchgeführt. Das bedeutet, es ändert sich eine Eigenschaft vom Typ `Double`, in diesem Fall die Eigenschaft `FontSize` des `Button`. Das Element, dessen Eigenschaft geändert werden soll, wird dabei über das Attribut `Storyboard.TargetName` beschrieben, während die zu ändernde Eigenschaft über das Attribut `Storyboard.TargetProperty` gesetzt wird.

Weitere Attribute unserer `DoubleAnimation` sind `From`, `To` und `Duration`. Dabei werden der Startwert der Eigenschaft, der Zielwert und die Dauer der Animation definiert.

Abbildung 8.5 und Abbildung 8.6 zeigen das Window zu Beginn und am Ende der Animation.

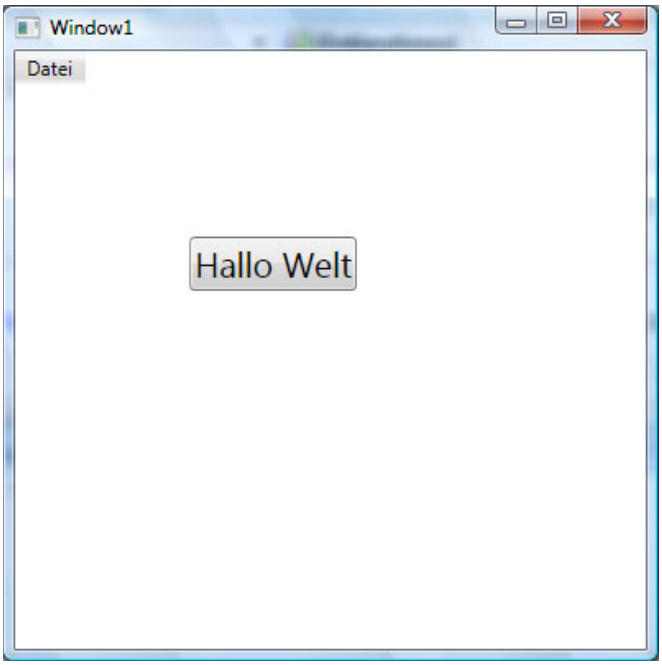


Abbildung 8.5
Window zu Beginn der Animation

Abbildung 8.6
Window am Ende der Animation

9

ADO.NET

Daten sind in vielerlei Hinsicht eine wichtige Sache und in vielen Applikationen geht es nur um die Präsentation von Daten und die Manipulation dieser Daten. Und genau an dieser Stelle kommen Datenbanken ins Spiel. Datenbanken persistieren jede mögliche Art von Daten und stellen diese jederzeit für eine personalisierte Ausgabe bereit. In diesem Kapitel beschäftigen wir uns damit, wie Sie mit ADO.NET Daten selektieren, präsentieren und eventuell geänderte Daten wieder zurückschreiben können. Als Datenbanksystem für die Beispiele wird der SQL Server 2008 von Microsoft verwendet.

Das Thema ADO.NET könnte locker ein ganzes Buch füllen. Eine der größten Aufgaben eines Entwicklers ist es, sich oftmals mit Datenbanken und deren Verbindungen auseinanderzusetzen. Microsoft hat enorme Veränderungen von DAO über RDO zu ADO und danach zu ADO.NET 1.1 und 2.0 vollzogen, so dass Sie, als Entwickler, sich mit immer neuen Möglichkeiten des Zugriffs auseinandersetzen müssen. Nicht nur die unterschiedlichen Datenbanksysteme, sondern auch die verschiedenen Möglichkeiten und Wege, wie Informationen aus der Datenbank geholt und gespeichert werden können, gilt es zu kennen.

Eines der Hauptziele bei der Realisierung von ADO.NET 4.0 war seitens Microsoft, dass alle Funktionen von früheren ADO.NET-Versionen weiterhin funktionieren, es wurde lediglich Funktionalität erweitert. Assistenten und die sogenannten SmartTags helfen Ihnen, ohne auch nur eine Zeile Code zu produzieren, erstaunlich effizienten Zugriff auf eine Datenbank zu erhalten. Sehr oft möchten aber viele Programmierer ihren Programmcode gerade bei Datenbankanwendungen selbst schreiben und auch dies wird Ihnen in diesem Kapitel gezeigt.

Lernen Sie in diesem Abschnitt zum einen, wie Sie auf Datenbanken zugreifen und Informationen erlangen, ohne viel Code zu schreiben, und zum anderen, wie Sie umfangreichere Datenbankanwendungen mit richtigem Code schreiben.

Der letzte Abschnitt dieses Kapitels wird sich mit dem immer wichtiger werdenden Thema XML beschäftigen.

Aber beschäftigen wir uns vielleicht als Erstes mit der Frage, was überhaupt ADO.NET ist.

9.1 Was ist ADO.NET?

ADO.NET ist ein Teil des .NET Frameworks, das eine Schicht bereitstellt, um mit unterschiedlichen Datenquellen zu kommunizieren. Eine Datenquelle kann dabei eine dateibasierte Datenquelle (Access-Datenbank, XML-Datei, SQL Express-Datenbank etc.) oder auch ein auf einem Server liegendes Datenbankmanagementsystem (SQL-Server, Oracle, DB2 etc.) sein. ADO.NET stellt uns dabei Tools und Objekte zur Verfügung, um diesen Zugriff sehr komfortabel gestalten zu können und damit sich der Entwickler keine allzu großen Gedanken über spezifische Implementierungen der verschiedenen Datenquellen machen muss.

ADO.NET ist somit unsere Programmierschnittstelle zu Datenbanken.

Sämtliche Klassen für den Datenzugriff finden wir im Namespace `System.Data`.

Der größte Unterschied von ADO.NET zu seinen Vorgängertechnologien DAO, RDO und ADO besteht darin, dass es sich bei ADO.NET um ein verbindungsloses Konzept handelt. Das bedeutet, dass eine Verbindung zur Datenquelle nur dann besteht, wenn tatsächlich gerade mit der Datenbank kommuniziert wird.

Das Szenario sieht in der Regel so aus, dass bestimmte Daten von der Datenquelle selektiert und lokal im Hauptspeicher verwaltet werden. Für das Selektieren der Daten besteht eine Verbindung zu unserer Datenquelle. Sobald die Daten jedoch in den Hauptspeicher übertragen wurden, wird diese Verbindung auch sofort wieder abgebaut. Der Anwender kann jetzt die Daten auswerten, aber auch ändern. Die Änderung geschieht jedoch erst im Speicher der Applikation. Erst wenn die Daten explizit in die Datenquelle zurückübertragen werden, wird wieder eine Verbindung zu unserem Datenbanksystem aufgebaut und die gewünschten Änderungen werden an die Datenbank übertragen. Ist dies geschehen, wird die Verbindung auch wieder abgebaut.

In den Vorgängerversionen wurde oftmals so programmiert, dass beim Programmstart die Verbindung zur Datenbank aufgebaut und beim Programmende wieder abgebaut wurde, dies war ein verbindungsorientiertes Konzept.

Da sich die Art und Weise, wie Applikationen entwickelt werden, in den letzten Jahren sehr stark verändert hat, wurde es auch immer wichtiger, für den Datenbankzugriff ein verbindungsloses Konzept zur Verfügung zu haben. Webapplikationen und verteilte Anwendungen bieten die Möglichkeit, dass viele Tausend Benutzer gleichzeitig Ihre Applikation nutzen, und somit wird es auch immer wichtiger, eine skalierbare Applikation zur Verfügung zu stellen. Verbindungslose Datenbankkonzepte bieten im Gegensatz zu verbindungsorientierten Technologien genau diese Skalierbarkeit.

Info

Der Anteil von Webanwendungen im Verhältnis zu herkömmlichen Windows-Applikationen ist seit Beginn dieses Jahrtausends deutlich gestiegen. Als ADO vor mittlerweile fünfzehn Jahren auf den Markt kam, gab es nur sehr, sehr wenige webbasierte Applikationen.

Ein weiterer wichtiger Gedanke bei der Implementierung von ADO.NET war die Zusammenarbeit mit dem immer wichtigeren und auch immer weiter verbreiteten Dateiformat XML. Wir werden im Laufe dieses Kapitels sehen, wie eng die Datenverarbeitung mittels ADO.NET mit XML zusammenliegt und wie einfach Daten nach XML serialisiert werden können.

Nachdem wir nun einen groben Überblick haben, was ADO.NET bedeutet, betrachten wir jetzt erst einmal, welche Objekte uns im Namensraum `System.Data` überhaupt zur Verfügung stehen.

Die Abkürzung für ADO.NET ist im Übrigen ein wenig irreführend. ADO steht nämlich für ActiveX Data Objects, aber mit ActiveX hat die ganze Sache eigentlich gar nichts mehr zu tun. Dieser Name wurde lediglich gewählt, um den Bezug zur Vorgängertechnologie nicht zu verlieren.

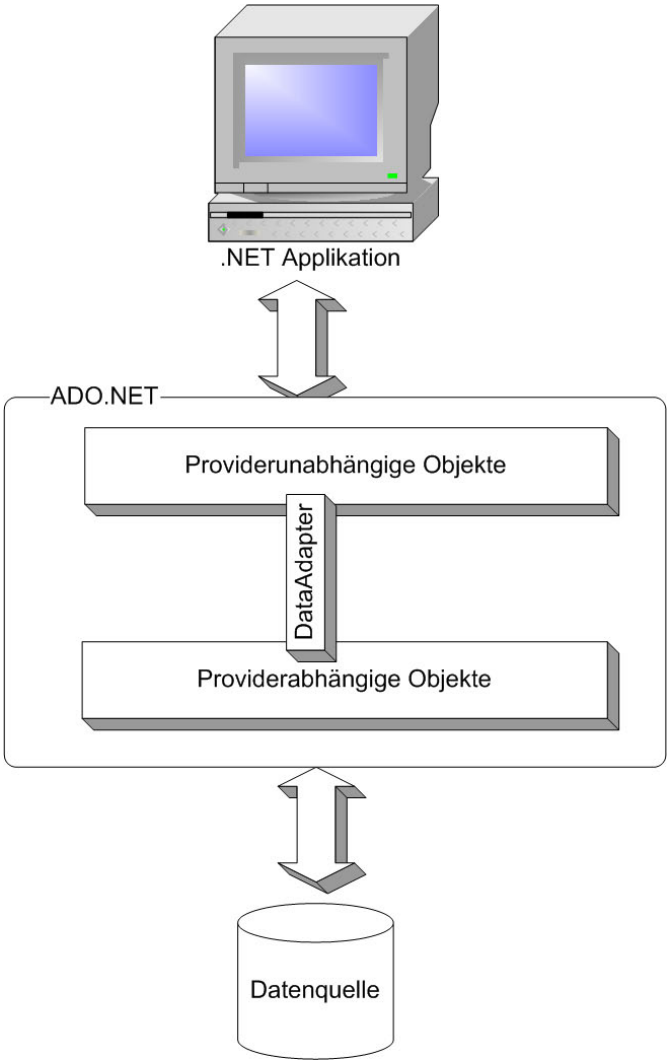
Auch wenn ADO.NET auf einem komplett anderen Konzept aufbaut, so wurden, soweit möglich, bekannte Objekte aus der Vorgängerversion übernommen. Erfahrene ADO-Entwickler werden deshalb einige bekannte Objekte wiederfinden, auch wenn diese mittlerweile anders funktionieren.

9.2 ADO.NET-Objekte

Bei der Betrachtung der .NET-Objekte sollten wir diese erst einmal in zwei unterschiedliche Kategorien aufteilen; zum einen in providerabhängige Objekte, die abhängig von einer speziellen Implementierung für eine spezifische Datenbank sind und auch mit der entsprechenden Datenquelle kommunizieren. Auf der anderen Seite haben wir die providerunabhängigen Objekte, die völlig unabhängig von den darunterliegenden Datenquellen sind.

Abbildung 9.1 zeigt das Zusammenspiel der providerabhängigen und providerunabhängigen Objekte.

Abbildung 9.1
Zusammenspiel der ADO.NET-Objekte



9.2.1 Providerabhängige Objekte

Innerhalb des Namespace `System.Data` gibt es weitere Namespaces, in denen Objekte für bestimmte verwaltete Provider liegen. Standardmäßig werden bei der Installation von Visual Studio 2010 sechs unterschiedliche managed Provider für Datenbankzugriffe mitinstalliert. In Tabelle 9.1 sehen Sie eine Auflistung der mitgelieferten Provider.

Namespace	Unterstützte Datenquellen
System.Data.SqlClient	Microsoft SQL Server ab der Version 7
System.Data.OracleClient	Oracle ab der Version 8.1.6 (gilt aber als veraltet; es wird empfohlen, den Provider von Oracle zu verwenden)
System.Data.Odbc	Für jegliche ODBC-Datenquelle
System.Data.OleDb	Für jegliche OleDb-Datenquelle (wie zum Beispiel Microsoft Access)
Microsoft.SqlServerCe.Client	Für SQL Server CE-Datenbanken
System.Data.SqlServerCe.3.5	Für SQL Server Compact- Datenbanken

Tabelle 9.1
Verwaltete Provider

Für viele weitere Datenbanksysteme werden von den jeweiligen Herstellern lizenzfreie Provider zur Verfügung gestellt (wie zum Beispiel MySQL, DB2 oder Cache).

Sämtliche providerabhängigen Objekte besitzen alle ein Präfix vor dem Objektnamen. Wenn ich gleich von einem `Connection`-Objekt spreche, dann bedeutet dies, dass das entsprechende `Connection`-Objekt im Namespace `System.Data.SqlClient` `SqlConnection` heißt, während es im Namespace `System.Data.OracleClient` ein `OracleConnection`-Objekt gibt. Die beiden Präfixe für den ODBC- und OleDb-Namensraum sind sinngemäß ODBC und OleDb.

Kommen wir aber nun zu den wichtigsten providerabhängigen Objekten:

Connection

Mit diesem Objekt wird eine Verbindung zu einer Datenquelle aufgebaut. Viele .NET-Datenprovider unterstützen dabei Connection-Pooling. Das bedeutet, dass Sie mehrere physikalische Datenbankverbindungen verwalten. Ein Öffnen eines `Connection`-Objekts ist somit nicht zwingend ein physikalisches Öffnen einer Datenbankverbindung, was einen zeitaufwändigen Prozess darstellen würde, sondern lediglich das Anfordern einer entsprechenden Ressource. Genauso ist das Schließen eines `Connection`-Objekts das Freigeben der entsprechenden Ressource.

DataAdapter

Dieses Objekt besteht intern aus vier weiteren `Command`-Objekten: einem Select-, Insert-, Update- und Delete-Command. Dieses Objekt kommuniziert mit dem providerunabhängigen `DataSet`, zu dem wir gleich kommen. Es befüllt das entsprechende `DataSet` mit Daten aus der Datenbank und persistiert die im `DataSet` durchgeführten Änderungen zurück in die Datenquelle. Beim Aufruf der Methoden eines `DataAdapter` brauchen Sie sich um den Verbindungsaufbau und -abbau der zugehörigen Datenquelle nicht zu kümmern.

Command

Dieses Objekt führt beliebige Anweisungen gegen die zugrunde liegende Datenquelle aus. Im Gegensatz zum `DataAdapter` müssen Sie sich beim Ausführen

von Befehlen selbst um den Verbindungsaufbau und -abbau der zugehörigen Datenquelle kümmern. Ein `Command`-Objekt kann dabei Daten selektieren, Daten manipulieren und an Transaktionen teilnehmen. Der Befehlstext kann dabei eine gültige SQL-Anweisung oder auch den Namen einer Stored Procedure darstellen.

Parameter

Parameterobjekte sind sehr eng mit `Command`-Objekten, auch denen innerhalb eines `DataAdapter`, verbunden. Anstatt bei den `Command`-Objekten Variablenwerte innerhalb der SQL-Anweisung durch das dynamische Zusammensetzen von Strings zu setzen, können in den Befehlstexten `Parameter` definiert werden. Diese `Parameter` werden dann später an die Parameterauflistung des entsprechenden `Command`-Objekts angehängt.

DataReader

Der `DataReader` entspricht in etwa einem schreibgeschützten, Forward-only-Cursor. Mit einem `DataReader` kann man sehr schnell Daten aus einer Datenquelle abrufen. Die Daten können jedoch nur ein einziges Mal lesend vom ersten bis zum letzten Datensatz durchlaufen werden. Ein `DataReader` kann aus einem `Command`-Objekt heraus erstellt werden und benötigt eine offene `Connection` exklusiv.

Transaction

Das `Transaction`-Objekt kann mehrere einzelne Anweisungen zu einer einzelnen untrennbaren Anweisung zusammenfassen. Entweder werden alle Anweisungen gegen die Datenbank erfolgreich durchgeführt oder keine einzige. Transaktionen laufen dabei atomar, konsistent, isoliert und dauerhaft ab. In ADO.NET sind nun auch verteilte Transaktionen über verschiedene Datenbanken hinweg möglich. Die zugehörige Funktionalität hierfür liegt im Namespace `System.Transactions` in der gleichnamigen Assembly.

9.2.2 Interfaces für die providerabhängigen Objekte

Obwohl in den einzelnen Namespaces tatsächlich unterschiedliche Objekte vorhanden sind, basieren alle providerabhängigen auf denselben Interfaces. Dies hat den Vorteil, dass die Funktionalität in den einzelnen Klassen auf das dahinterliegende Datenbanksystem optimiert ist, jedoch alle Klassen dieselben Methoden, Eigenschaften und Ereignisse besitzen. Somit ist der Umstieg auf ein anderes Datenbanksystem, zumindest aus der Sicht von ADO.NET, eine sehr einfache Sache, die keinerlei Umlernen erfordert.

Die betreffenden Interface-Definitionen liegen dabei im Namespace `System.Data`. Die Interfaces heißen:

- IDbConnection
- IDbDataAdapter
- IDbCommand
- IDbParameter

- IDbDataReader
- IDbTransaction

Auf diese Art und Weise ist gewährleistet, dass sämtliche gleichartigen Objekte kompatible Schnittstellen besitzen, was vor allem bei der Nutzung der in ADO.NET neu implementierten `DbProviderFactories` wichtig ist.

Für den Zugriff auf Microsoft SQL Server- oder Oracle-Datenbanken stehen Ihnen ja mit `System.Data.SqlClient` beziehungsweise `System.Data.OracleClient` eigene Namensräume mit optimierten Klassen zur Verfügung. Sie können natürlich auch auf diese Datenbanksysteme mit `System.Data.OleDb` oder `System.Data.Odbc` zugreifen. Seien Sie sich jedoch der Tatsache bewusst, dass Sie dabei einiges an Leistung verlieren und auch datenbankspezifische Datentypen in diesen beiden sehr allgemein gehaltenen Namensräumen nicht zur Verfügung stehen. Vor allem im Namespace `System.Data.SqlClient` finden Sie einige zusätzliche Schnittstellen, welche die Funktionalität für den Zugriff auf den SQL Server deutlich erweitern. Die Beschreibung dieser Zusatzfunktionalitäten ist auch Bestandteil dieses Kapitels.

9.2.3 Providerunabhängige Objekte

Innerhalb des Namespace `System.Data` liegen direkt die providerunabhängigen Objekte, die mit den providerabhängigen Objekten kommunizieren können, wie Sie in Abbildung 9.1 sehen können.

DataSet

Das `DataSet` ist das wichtigste providerunabhängige Objekt in ADO.NET, denn es ist die zentrale Stelle, in der die Daten innerhalb Ihrer Applikation gespeichert werden. Ein `DataSet` ist dabei eine Auflistung von Datentabellen, die untereinander in Beziehung stehen können. Ein `DataSet` wird durch einen `DataAdapter` befüllt, hat jedoch keinerlei Wissen über die Herkunft der Daten. Wenn in Ihrer Anwendung Änderungen durchgeführt werden, so werden diese zuerst ins `DataSet` geschrieben, das heißt, die Daten werden lediglich im Hauptspeicher geändert und nicht in der Datenbank. Zum Zurückschreiben zur Datenbank ist wiederum ein `DataAdapter` nötig.

DataTable

Wie eben bereits erwähnt, ist ein `DataSet` eine Auflistung von Datentabellen, die aus verschiedenen, auch unterschiedlichen Datenquellen befüllt werden können. Eine `DataTable` beinhaltet die Daten, die ein `SelectCommand` eines `DataAdapter` aus einer Datenquelle liest. Eine `DataTable` kann mittlerweile unter .NET auch direkt befüllt werden, ohne ein `DataSet` nutzen zu müssen.

DataRow

Eine `DataRow` repräsentiert eine Zeile innerhalb einer `DataTable`. Für viele Programmierer wird es sehr ungewöhnlich sein, sich ohne Move-Operationen durch die Zeilen einer Tabelle zu bewegen. Eine `DataTable` in ADO.NET besitzt eine `Rows`-Auflistung, durch die mit einer `For Each`-Schleife iteriert werden kann.

DataColumn

Genauso wie eine Tabelle eine Auflistung von Zeilen besitzt, hat die Tabelle auch eine Auflistung von Spalten. Eine `DataColumn` ist dabei eine einzelne Spalte innerhalb einer Tabelle. Auch die Spaltenauflistung kann mit `For Each` durchlaufen werden.

DataView

Mit einer `DataView` können Sie sich eine bestimmte Sicht auf eine `DataTable` erstellen. Mit einer `DataView` können Sie dabei Daten sortieren oder filtern.

DataRelation

Da die Daten innerhalb eines `DataSet` kein Wissen über ihre Herkunft haben, können sie auch nicht wissen, dass ursprünglich innerhalb der Datenquelle eine Relation zwischen zwei Tabellen besteht. Mit einem `DataRelation`-Objekt kann man solche Beziehungen im `DataSet` nachbilden.

So, nachdem wir nun die wichtigsten Objekte einmal kurz angesprochen haben, wollen wir uns anschauen, wie wir mit diesen Objekten arbeiten können.

In den folgenden Abschnitten wollen wir eine Windows-Applikation erstellen, in der wir auf eine SQL Server-Datenbank zugreifen und die wichtigsten Objekte demonstrieren. Als Datenbank verwende ich dabei die Microsoft-Beispieldatenbank *Northwind*.

Zuerst will ich Ihnen zeigen, wie Sie mit Assistenten und Designern und somit mit nur sehr wenig Programmcode Datenbindung durchführen können. Im zweiten Schritt geht es dann darum, mit reiner Programmierung, ohne die Hilfe von Designercode, die Datenbankzugriffe zu realisieren.

Für welche Art Sie sich letztendlich entscheiden, hängt meistens davon ab, welche Art von Applikation Sie entwickeln wollen. Bei Systemen mit einem einzigen Benutzer, also ohne konkurrierende Zugriffe, werden die Designer ihre Arbeit wohl zu Ihrer vollsten Zufriedenheit durchführen. Wenn jedoch mehr Benutzer gleichzeitig auf dieselben Datenquellen zugreifen, wird wohl die Tendenz eher dahin gehen, sich ein bisschen mehr Mühe zu machen und die Zugriffe selbst in die Hand zu nehmen.

9.3 Databinding mit Designer

War Datenbindung bereits in den ersten .NET-Versionen eine teilweise sehr einfache Sache, so hat Microsoft in dieser Version in Bezug auf Komfort noch eins draufgesetzt. Komplette Anwendungen können fast ausschließlich mit Drag&Drop realisiert werden. Die in Werbekampagnen versprochenen 70% weniger Code für Standardaufgaben bei Datenbankzugriffen wurden nach meiner Einschätzung tatsächlich eingehalten.

Ein Punkt, der ebenso zum Thema Datenbindung gehört, sind die Steuerelemente, die für Datenbindung verwendet werden können. Prinzipiell ist fast jedes Steuerelement für Datenbindung geeignet, was ich Ihnen auch noch demonstrieren werde. Doch gibt es auch spezielle Steuerelemente, die ausschließlich für die Datenbindung verwendet werden. Das wichtigste hiervon ist das `DataGridView`, der Nachfolger des `DataGrid` aus der .NET Framework-Version 1.1.

Bevor wir aber zu unserer kleinen Beispielanwendung kommen, schaffen wir uns erst noch einen Überblick über Datenbindung.

9.3.1 Übersicht

Datenbindung kann in ADO.NET grundsätzlich auf zwei verschiedenen Wegen funktionieren: einmal unter der Zuhilfenahme von sogenannten Datenquellen und einmal mit einer direkten Bindung. Dabei würde die Datenquelle nur den direkten Zugriff kapseln.

Die Daten für die Datenbindung können dabei direkt aus einer Datenbank kommen und über ein `DataSet` oder einen `DataReader` an eine Datenquelle beziehungsweise direkt an ein Steuerelement gebunden werden. Hierbei ist zu erwähnen, dass der `DataReader` nur für Datenbindung in Webapplikationen genutzt werden kann, jedoch nicht in Windows-Applikationen. Ebenso könnte ein `DataSet` auch als Rückgabewert von einer Webservice-Methode genutzt werden, so dass ein WebService auch eine gültige Datenquelle sein kann.

Mit ADO.NET ist es auch möglich, direkt Objekte aus Middleware-Klassen zur Datenbindung zu verwenden. Dabei können bestimmte Methoden dieser Klasse zum Befüllen der Daten sowie zum Zurückschreiben der Daten in das Objektmodell genutzt werden. Abbildung 9.2 soll die verschiedenen Datenbindungsmöglichkeiten illustrieren.

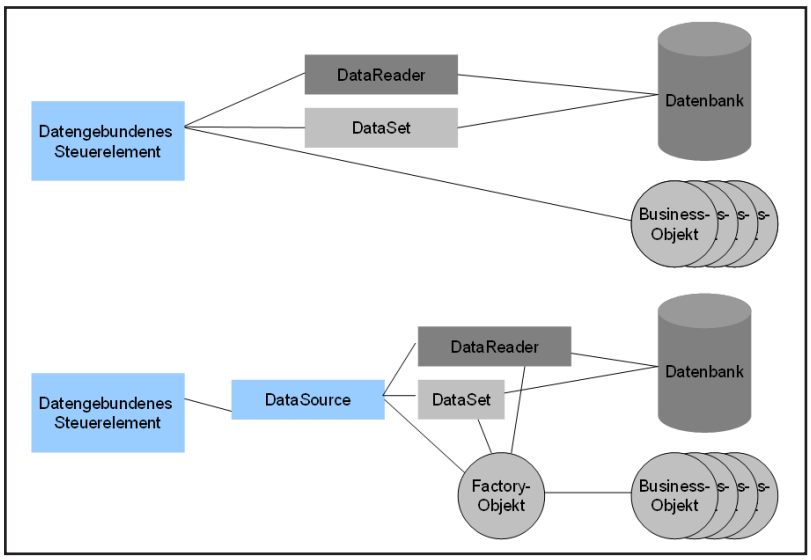


Abbildung 9.2
Datenbindung

So, nun wollen wir aber zu unserer kleinen Beispielanwendung kommen, und wir dürfen schon gespannt sein, wie viele Zeilen Programmcode wir tatsächlich für die Anwendung benötigen.

9.3.2 Beispielanwendung

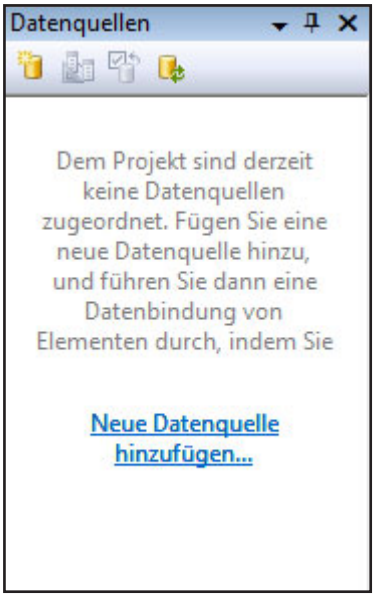
In dieser Anwendung wollen wir Schritt für Schritt möglichst viele der oben beschriebenen Objekte und Möglichkeiten der Datenbindung mit unterschiedlichen Steuerelementen kennenlernen.

In dem Beispiel wird auf die Beispieldatenbank Northwind zugegriffen. Falls Sie diese Datenbank nicht haben, können Sie sich diese unter *http://www.microsoft.com/downloads/en/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en* downloaden und in Ihrem SQL-Server installieren.

Nachdem wir ein neues Projekt vom Typ Windows Forms-Anwendung angelegt haben, wollen wir uns als Erstes eine eigene Datenquelle anlegen. Dazu müssen wir zunächst das Datenquellenfenster sichtbar machen. Falls die Datenquellen noch nicht angezeigt werden, können Sie das Fenster sehr komfortabel über das Menü Daten – Datenquellen anzeigen sichtbar machen.

Nachdem wir bislang noch keine Datenquelle haben, sehen wir im Datenquellenfenster, siehe Abbildung 9.3, nur einen Link zum Hinzufügen einer neuen Datenquelle.

Abbildung 9.3
Datenquellenfenster



Nachdem wir auf den Link klicken, erscheint der nächste Dialog, wie in Abbildung 9.4 dargestellt, in dem wir gefragt werden, welchen Typ unsere Datenquelle haben soll. Wir wählen hier den Eintrag Datenbank aus.

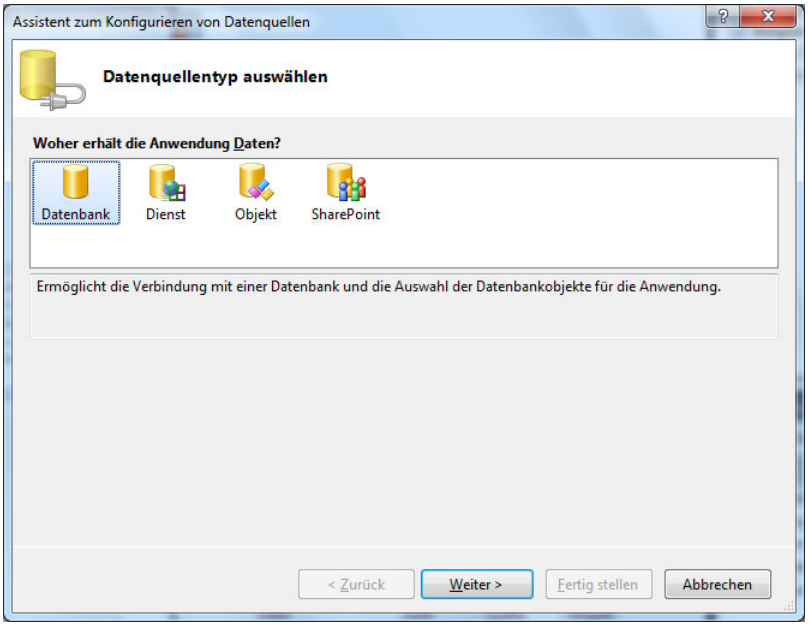


Abbildung 9.4
Auswahl des Datenquellentyps

Daraufhin müssen Sie im nächsten Dialog das Datenbankmodell auswählen. Wählen Sie dazu den Eintrag DataSet aus. Mehr zur Alternative Entity Data Model erfahren Sie im nächsten Kapitel.

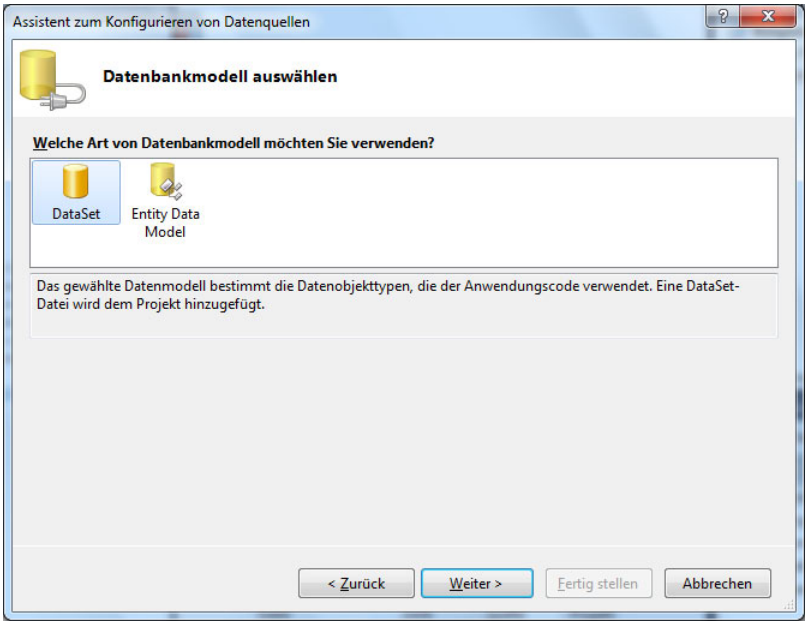
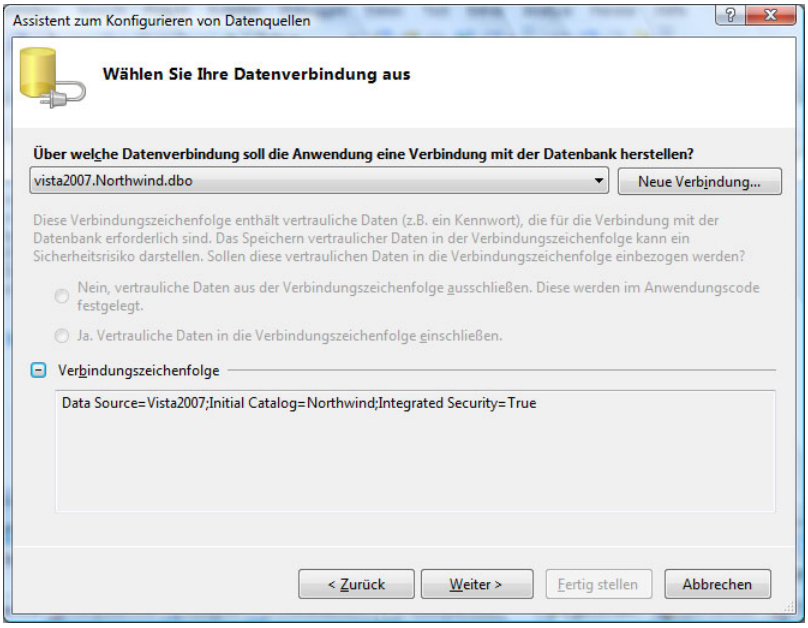


Abbildung 9.5
Auswahl des Datenbankmodells

Im folgenden Dialog, den Sie in Abbildung 9.6 sehen, muss nun die gewünschte Datenbank ausgewählt werden, was in der Abbildung schon geschehen ist. Durch den Button Neue Verbindung können Sie eine Verbindung zu einer beliebigen Datenbank auswählen.

Abbildung 9.6
Auswahl der Datenbankverbindung



Im Dialog Verbindung hinzufügen, siehe Abbildung 9.7, wählen Sie zuerst die entsprechende Datenquelle aus. In unserem Beispiel lassen wir den Standardeintrag *Microsoft SQL Server* stehen. Mit dieser Datenquelle können Sie auf SQL Server-Datenbanken ab der Version 7 zugreifen sowie auf den SQL Server Express 2008. Wollen Sie ein anderes Datenbanksystem auswählen, können Sie das über die Schaltfläche Ändern... tun.

In dem Kombinationsfeld Servername können Sie nun entweder auf den lokalen Server, die Standardinstanz oder auch eine andere Instanz zugreifen oder auch einen SQL Server auf einem anderen Rechner innerhalb Ihres Netzwerks auswählen. Zur Auswahl der Standardinstanz auf dem lokalen SQL Server geben Sie ganz einfach `(local)` oder auch `localhost` ein.

Danach können Sie die Anmeldeinformationen angeben, wobei bei der Windows-Authentifizierung die Anmeldeinformationen des angemeldeten Windows-Benutzers verwendet werden. Bei der Auswahl von SQL Server-Authentifizierung müssen Sie einen Benutzernamen und ein Kennwort angeben.

Im Abschnitt Mit Datenbank verbinden wählen Sie letztendlich noch die Datenbank aus, mit der Sie sich verbinden wollen. Wenn Sie den SQL Server Express verwenden, dann wählen Sie unter Datenbankdatei anhängen die entsprechende Datei aus, mit der Sie sich verbinden wollen.

SQL Server Express ist ein dateibasiertes Datenbanksystem.

Info

Mit der Schaltfläche Testverbindung können Sie dann noch überprüfen, ob alle Ihre Angaben korrekt sind.

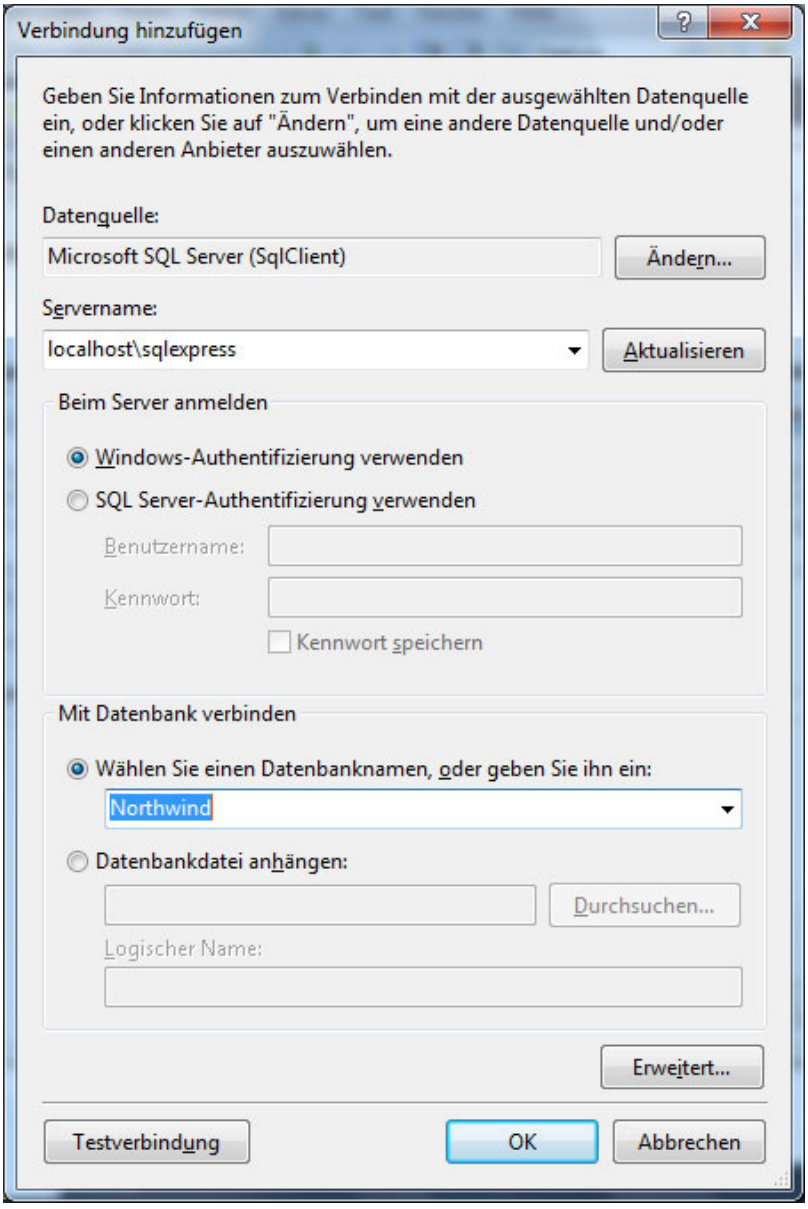


Abbildung 9.7
Auswahl einer Datenbank

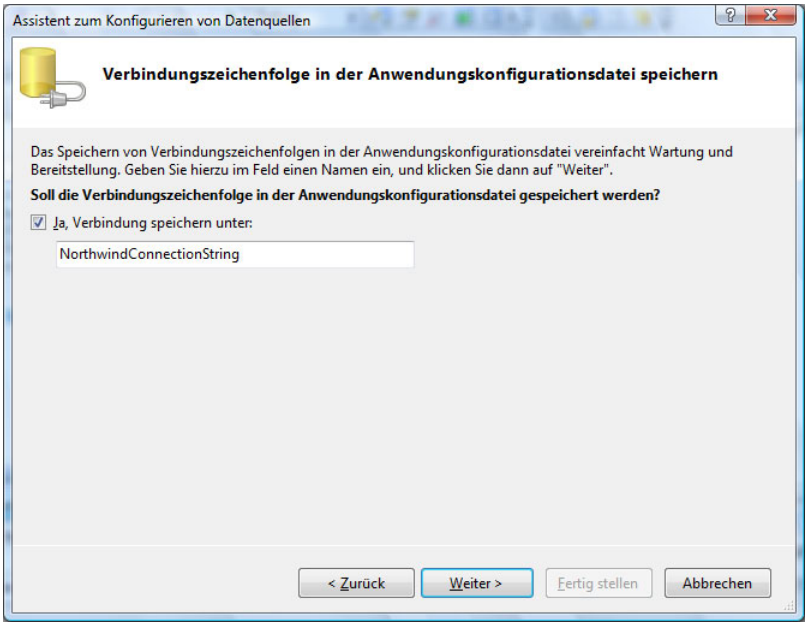
Wenn Sie mit der Schaltfläche OK Ihre Angaben bestätigen, kommen Sie zum ursprünglichen Dialog zurück. Sie können sich dann die generierte Verbindungszeichenfolge anzeigen lassen, indem Sie auf das +-Zeichen vor VERBINDUNGSZEICHENFOLGE klicken, wie in Abbildung 9.6 geschehen. Ihre Verbindungszeichenfolge wird sich gegenüber der Abbildung vermutlich darin unterscheiden, dass bei Ihnen unter Umständen `(local)` oder ein anderer Servername an der Stelle steht, wo in der Abbildung der Eintrag VISTA2007 vorhanden ist.

Die Verbindungszeichenfolge, die in Abbildung 9.6 dargestellt ist, wird dabei in die applikationsspezifische Konfigurationsdatei eingetragen, falls Sie diese Option im darauf folgenden Dialog nicht deaktivieren.

Listing 9.1 connectionStrings-Ausschnitt der applikationsspezifischen Konfigurationsdatei

```
<connectionStrings>
  <add
    name="BeispielanwendungDesigner.My.
      MySettings.NorthwindConnectionString"
    connectionString="Data Source=localhost\SqlExpress;
      Initial Catalog=Northwind;
      Integrated Security=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

Abbildung 9.8 Speicherung der Verbindungszeichenfolge



Nachdem alle Angaben zur Datenbank gemacht wurden, werden Informationen über Tabellen, Sichten, Stored Procedures und Functions aus der Datenbank abgerufen. Welche Daten davon in Ihrer Applikation angezeigt werden, können Sie im folgenden Dialog DATENBANKOBJEKTE AUSWÄHLEN angeben.

Für unser Beispiel wählen wir, wie in Abbildung 9.9 gezeigt, die Tabellen *Customers*, *Orders*, *Order Details* und *Products* aus.

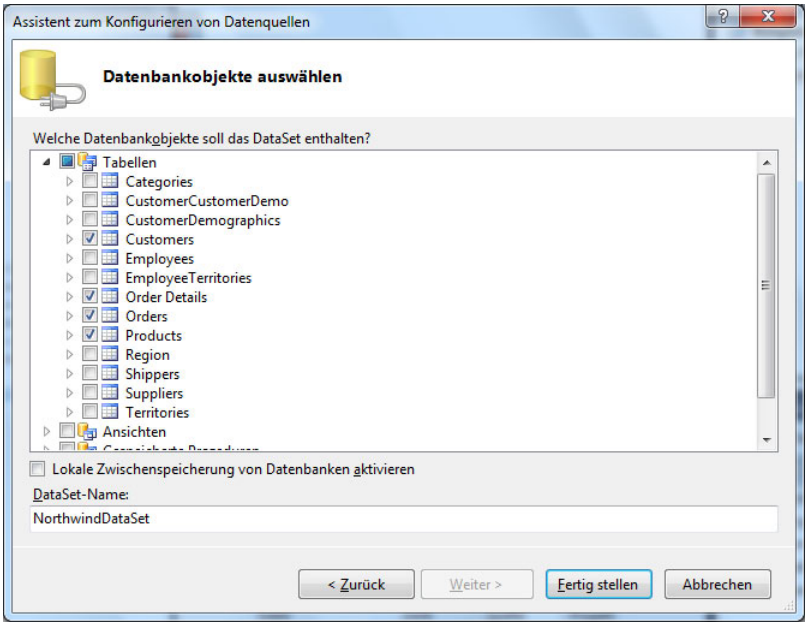


Abbildung 9.9
Tabellenauswahl

Mit der Schaltfläche FERTIG STELLEN beenden Sie das Hinzufügen einer neuen Datenquelle. Die Datenquelle besteht dabei aus den vier gewählten Tabellen.

Sie haben natürlich noch die Möglichkeit, weitere Datenquellen hinzuzufügen oder bestehende Datenquellen zu konfigurieren.

Tatsächlich wurde jetzt ein Objekt mit dem Namen `NorthwindDataSet` vom Typ `DataSet` angelegt, das intern aus vier `DataTables` *Customers*, *Orders*, *Order Details* und *Products* besteht.

Um nun noch Beziehungen zwischen den Tabellen zu definieren, können Sie das `DataSet` bearbeiten, indem Sie im Kontextmenü des Datenquellenfensters den Eintrag DATASET MIT DESIGNER BEARBEITEN anklicken.

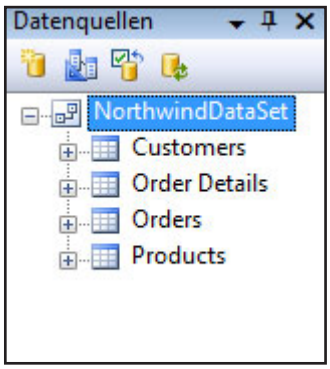


Abbildung 9.10
Konfigurierte Datenquelle

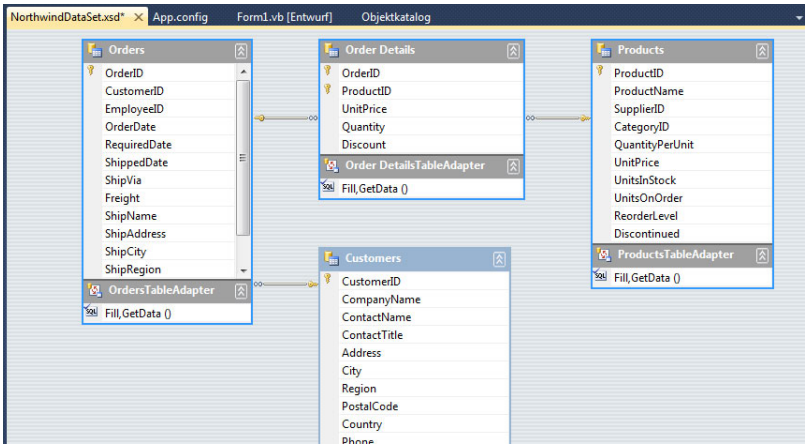


Abbildung 9.11
Relation zum DataSet hinzufügen

Wie Sie in Abbildung 9.11 sehen, hat der Designer bereits Beziehungen zwischen den Tabellen *Customers* zu *Orders* und von *Orders* zu *Order Details* sowie von *Order Details* zu *Products* erkannt. Wenn Sie weitere Beziehungen definieren wollen, könnten Sie im DataSetDesigner über das Kontextmenü HINZUFÜGEN eine weitere Relation zum `DataSet` hinzufügen. In unserem Beispiel fügen wir aber keine weitere Relation hinzu.

So, nun haben wir alles definiert, was wir an Daten anzeigen wollen. Im nächsten Schritt binden wir noch diese Daten an Steuerelemente innerhalb unseres Formulars.

Und dabei ist wieder das Datenquellenfenster unsere Schaltzentrale. Als Erstes wollen wir die Kunden (*Customers*) auf einem Formular darstellen, und zwar in einer Einzelansicht, durch die wir blättern können.

Sie können dabei auswählen, wie Sie in Abbildung 9.12 sehen, ob die Tabelle in einer `DataGridView`, also als tabellarische Ansicht der Daten, oder in einer Detailansicht dargestellt werden soll. Es kann für jede Spalte innerhalb der Tabelle auch noch definiert werden, welches Steuerelement für die Anzeige des Spaltenwertes gewählt werden soll. Datumsfelder werden zum Beispiel standardmäßig in einem `DateTimePicker` dargestellt. Spalten, die nicht angezeigt werden sollen, werden einfach mit dem Eintrag [KEINE] versehen.

Nachdem wir alle Spalten nach unseren Wünschen konfiguriert haben, können wir die Tabelle *Customers* ganz einfach auf ein Formular ziehen.

Dabei werden gebundene Steuerelemente für jede anzuzeigende Spalte auf einem Formular angelegt.

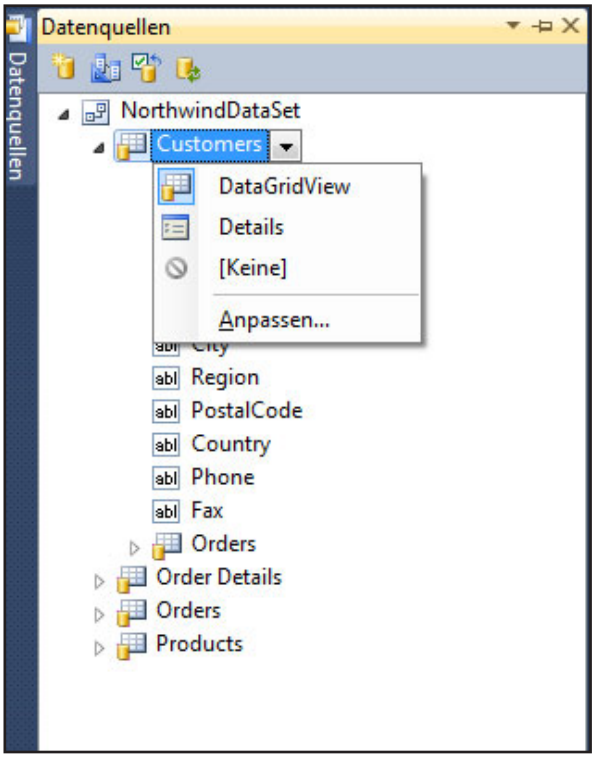


Abbildung 9.12
Auswahl der Anzeigeelemente

In Abbildung 9.13 sehen Sie das angelegte Formulardesign.

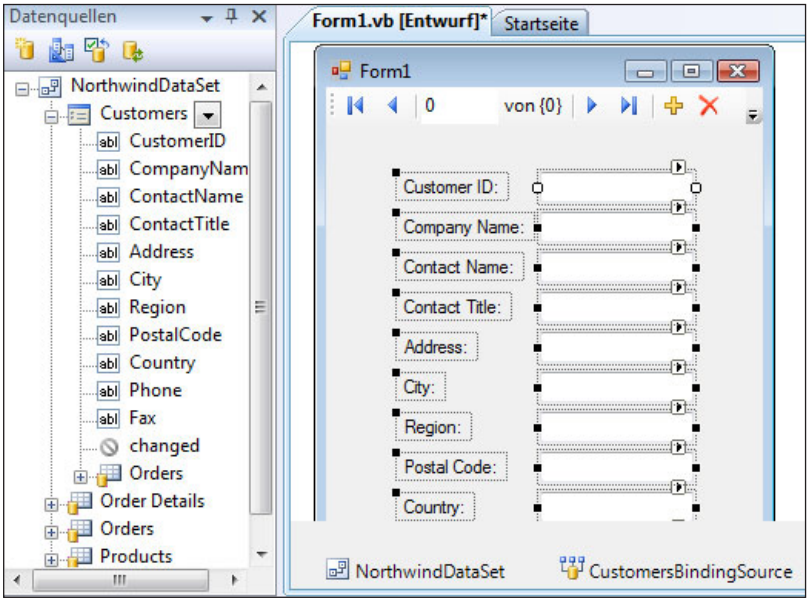


Abbildung 9.13
Automatisch angelegte gebundene Steuerelemente

Außerdem sehen Sie im sogenannten Komponentenfach fünf neue Objekte, die zu unserem Projekt hinzugefügt wurden.

- NorthwindDataSet

  Ein typisiertes DataSet, für das eine generische Klasse erstellt wurde. Innerhalb dieses DataSets sind sämtliche Tabellen- und deren Spalteninformationen enthalten. `NorthwindDataSet` ist dabei von der Basisklasse `DataSet` abgeleitet worden.

- CustomersBindingSource

  Eine BindingSource verwaltet Bewegungen innerhalb eines DataSet, um abhängige Tabellen zu synchronisieren. Wenn alle Daten, die angezeigt werden, auf dieselbe BindingSource zugreifen, dann synchronisieren sich die Anzeigen in den Tabellen automatisch.

- CustomersTableAdapter

  Dieser TableAdapter stellt sämtliche Funktionen zum Selektieren und Zurückschreiben der Daten in die Tabelle *Customers* zur Verfügung. Über den TableAdapter kann genau konfiguriert werden, wie die Daten zurückgeschrieben und selektiert werden. Wenn Sie sich genau anschauen wollen, was hinter so einem TableAdapter steckt, dann lassen Sie sich im Projektmappen-Explorer einfach alle Dateien anzeigen. Sie können dann den Eintrag *NorthwindDataSet.xsd* aufklappen. Mit einem Doppelklick können Sie dann die Datei *NorthwindDataSet.Designer.vb* öffnen. Innerhalb dieser generisch angelegten Codedatei gibt es unter anderem eine Definition für eine Klasse `CustomersTableAdapter`, in der sämtliche Funktionalität für den Zugriff auf die Tabelle *Customers* hinterlegt ist.

- CustomersBindingNavigator

  Der CustomersBindingNavigator ist ein Steuerelement, das auf unserem Formular angelegt wurde. Mit diesem BindingNavigator können Sie komfortabel durch den Datenbestand blättern, Datensätze löschen, Datenänderungen speichern und neue Daten hinzufügen. Welche Buttons dabei in dem BindingNavigator angezeigt werden, können Sie natürlich wieder selbst konfigurieren.

- TableAdapterManager

  Ein Manager für die Verwaltung von TableAdapter, in dem zum Beispiel die Reihenfolge definiert werden kann, in der die SQl-Statements an die Datenbank geschickt werden (z.B. InsertUpdateDelete) und ob die Inhalte der entsprechenden Datentabellen vor dem Befüllen gelöscht werden sollen.

Nun können wir bereits unser Programm starten und werden staunen, dass ohne eine einzige Zeile Programmcode schon so gut wie alles funktioniert.

Aber wir haben ja nicht noch weitere drei Tabellen in unserem `DataSet` hinzugefügt, um nur die Kundendaten zu betrachten. Im nächsten Schritt wollen wir noch die zugehörigen Bestellungen aus der Tabelle *Orders* anzeigen. Wir haben im Datenquellenfenster jetzt zweimal den Eintrag für die *Orders*-Tabelle; einmal direkt in der ersten Ebene und noch als weiteren Eintrag unter der Tabelle *Customers*.

Wenn Sie den Eintrag, der direkt unter dem `DataSet` hängt, auswählen, dann werden sämtliche Daten aus dieser Tabelle angezeigt, unabhängig vom ausgewählten Kontakt. Wenn Sie jedoch den Eintrag, der unter *Customers* steht, aufs Formular ziehen, dann werden nur Bestellungen für den ausgewählten Kunden angezeigt, wobei sich die Anzeige beim Blättern durch die Kunden automatisch aktualisiert. Genau das wollen wir und deswegen ziehen wir diesen Eintrag, der über die Beziehung definiert ist, als `DataGridView` auf unser Formular.

Unser Komponentenfach wird wiederum um zwei neue Objekte erweitert, einer `OrdersBindingSource` und einem `OrdersTableAdapter`, die dieselben Funktionen der entsprechenden `Customers`-Objekte übernehmen.

Wenn Sie nun bestimmte Spalten im `DataGridView` ausblenden oder anders konfigurieren wollen, dann können Sie dies machen, indem Sie im Eigenschaftsfenster die Schaltfläche für den Eintrag `Columns` anklicken oder über das Kontextmenü den Eintrag Spalten Bearbeiten... auswählen. Die Auswahlmöglichkeiten sehen Sie in Abbildung 9.14. Wieder können wir, ohne noch immer eine Zeile selbst programmiert zu haben, die Anwendung starten. Sobald durch die Kontaktdaten geblättert wird, passt sich die Anzeige in unserer `DataGridView` automatisch an.

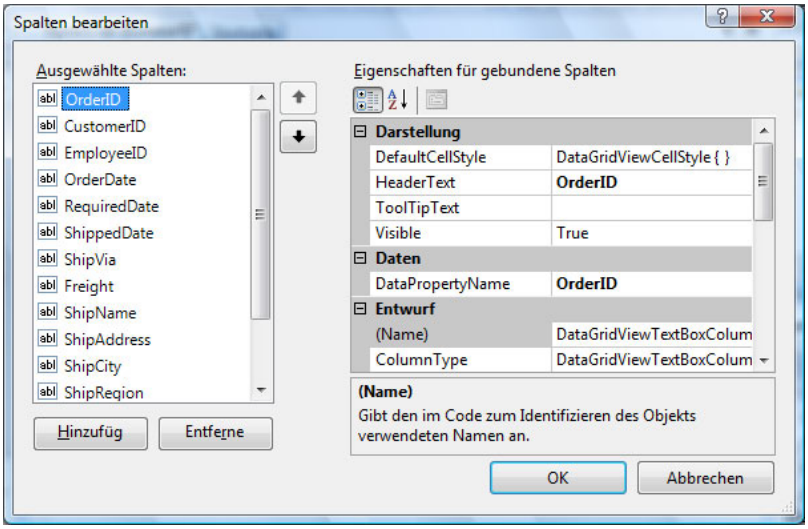


Abbildung 9.14
Spaltendefinitionen für eine DataGridView

Als Nächstes wollen wir noch die Bestelldetails anzeigen. Dazu ziehen wir den Eintrag *Order Details*, der unter dem bereits geschachtelten Eintrag *Orders* aufgelistet ist, auf unser Formular. Damit ist wiederum gewährleistet, dass die Daten aufgrund der im `DataSet` definierten Beziehungen korrekt angezeigt werden.

Wenn Sie jetzt die Anwendung starten, dann sehen Sie alle ausgewählten Daten. In den Bestelldetails sehen wir jedoch nur einen Fremdschlüssel für das bestellte Produkt. Viel schöner wäre es doch, wenn wir stattdessen die beschreibende Produktbezeichnung anzeigen könnten.

Öffnen Sie dafür noch einmal für das neue `DataGridView`, in dem die Bestelldetails angezeigt werden, den Dialog zum Bearbeiten der Spalten über die `Columns`-Auflistung des neuen Steuerelements.

In dem Dialog wählen Sie erst die Eigenschaften für die Spalte *ProductId* aus und ändern danach den *ColumnType* auf *DataGridViewComboBoxColumn*. Somit haben wir an der entsprechenden Stelle schon eine *ComboBox*. Jetzt müssen wir nur noch definieren, wo die Daten für die Liste herkommen. Das geben wir alles im Abschnitt Daten ein.

Dazu setzen wir die `DataSource` der `ComboBox` auf die `ProductBindingSource`, die wir aus einer Liste auswählen können. Dann setzen wir noch das `DisplayMember` auf die Spalte *ProductName* der Tabelle *Products* und das `ValueMember` auf *ProductId*, wie in Abbildung 9.15 zu sehen ist.

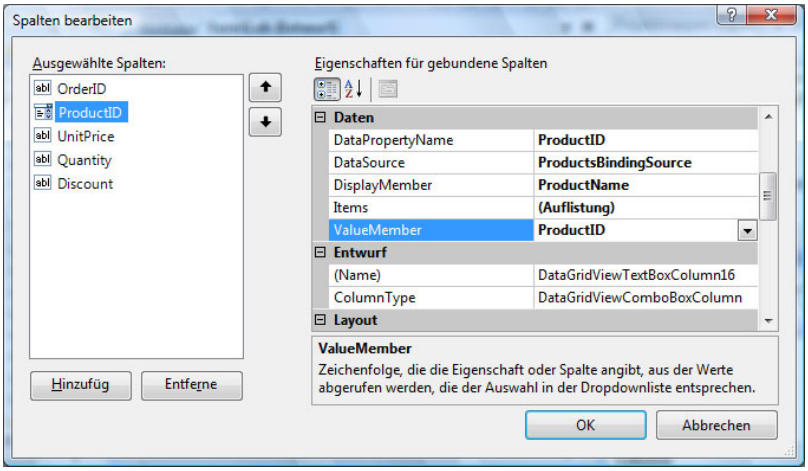


Abbildung 9.15
Einstellungen für die Spalte ProductId

Und schon ist ohne Code unsere Anwendung so gut wie fertig.

Nach dem Starten der Applikation sehen Sie alle gewünschten Daten und können auch durch diese Daten navigieren, wie Sie in Abbildung 9.16 sehen.

So schön diese Designerunterstützung zum Präsentieren von Daten auch sein mag, wenn es darum geht, die Daten letztendlich auch wieder in die Datenquelle zurückzuschreiben, verzichte ich eher auf die Nutzung des Designers und nehme die Sache selbst in die Hand. Denn lieber schreibe ich ein paar Zeilen Programmcode mehr, als dass ich mich bei Problemen dann durch 6272 Zeilen – so viele waren es bei unserem Beispiel – Designercode durchkämpfe. Meinen eigenen geschriebenen Code kenne ich zum einen viel besser und außerdem habe ich die Gewähr, dass meine Änderungen nicht durch den Designer wieder überschrieben werden.

Ich möchte jetzt die Designer nicht grundsätzlich verurteilen und um Daten zu präsentieren, greife ich auch sehr oft auf die Designerunterstützung zurück, aber gerade beim Speichern von Daten in einer Multibenutzeranwendung würde ich mich nicht auf sie verlassen wollen.

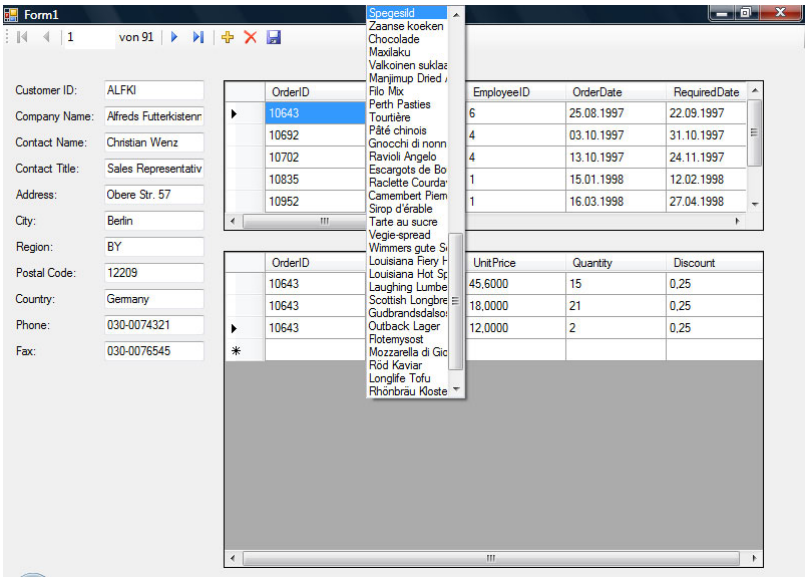


Abbildung 9.16
Screenshot der Applikation

Oftmals ist eine Mischung aus Designern und eigener Programmierung der beste Weg. Und das bringt uns zum nächsten Kapitel, das beschreibt, wie Sie mit den einzelnen Datenbankobjekten programmatisch umgehen können.

9.4 Programmierung

In diesem Kapitel erfahren Sie, wie Sie die wichtigsten Objekte programmatisch nutzen können. Dies will ich in kleinen Beispielprogrammen zeigen.

Info

Die Beispiele in diesem Abschnitt für providerabhängige Objekte sind mit den Objekten aus dem Namespace `System.Data.SqlClient` umgesetzt worden. Die äquivalenten Objekte in den anderen Namespaces besitzen dieselben beschriebenen Methoden und Eigenschaften.

In allen Formularen ist außerdem eine Importanweisung auf den Namespace `System.Data.SqlClient` gesetzt.

Beginnen wollen wir dabei mit einem Objekt, ohne das kein Datenbankzugriff möglich ist, dem `Connection`-Objekt.

9.4.1 SqlConnection

Mit dem `Connection`-Objekt kann eine Verbindung zu einer Datenquelle auf- und abgebaut werden.

Damit eine Verbindung geöffnet werden kann, muss man einen gültigen `ConnectionString` angeben. Der `ConnectionString` kann bereits dem Konstruktor übergeben werden, er kann aber auch explizit über die entsprechende Property gesetzt werden.

Tipp

Ein `Connection`-Objekt wird in einem Formular in verschiedensten Ereignishandlern benutzt. Deswegen empfiehlt es sich, ein `Connection`-Objekt als eine formularweite Variable zu definieren. Außerdem können Sie bei der Definition auch sofort den Konstruktor aufrufen und den entsprechenden `ConnectionString` angeben.

Als Nächstes stellt sich die Frage, ob man den `ConnectionString` wirklich hart codiert im Programmcode stehen haben will. Ich denke eher nicht, deswegen empfiehlt es sich, den `ConnectionString` in die applikationsspezifische Konfigurationsdatei auszulagern.

Den entsprechenden String können Sie dabei sehr einfach mit dem `ConfigurationManager`-Objekt auslesen.

Um das `ConfigurationManager`-Objekt nutzen zu können, müssen Sie Ihrem Projekt einen Verweis auf die Bibliothek *System.Configuration.dll* hinzufügen.

Der Eintrag in der *app.config*-Datei könnte wie folgt aussehen:

```
<connectionStrings>
  <add
    name="NorthwindConnection"
      connectionString="Data Source=(local)\SqlExpress;
      Initial Catalog=Northwind;
      Integrated Security=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

Tipp

Oftmals wissen Sie vielleicht nicht, wie der `ConnectionString` für eine bestimmte Datenbank auszusehen hat. Versuchen Sie es doch einfach unter www.connectionstrings.com.

Die Definition des `Connection`-Objekts sieht wie folgt aus:

```
Private con As New SqlConnection( _
    System.Configuration.ConfigurationManager. _
    ConnectionStrings("NorthwindConnection"). _
    ConnectionString)
```

Sie können jetzt an jeder beliebigen Stelle die Verbindung mit der Methode `Open()` öffnen beziehungsweise mit der Methode `Close()` auch wieder schließen. Wie bereits am Anfang dieses Kapitels erwähnt, handelt es sich bei ADO.NET um ein verbindungsloses Konzept, das bedeutet, dass Sie sämtliche Anweisungen, die Sie an die Datenbank schicken, mit `Open()` und `Close()` umschließen.

Wenn Sie sich nicht sicher sind (zum Beispiel innerhalb einer Fehlerbehandlung), ob Ihre Verbindung bereits geöffnet ist, können Sie die Eigenschaft `State` abfragen.

Hinter dieser Eigenschaft steht eine Enumeration von möglichen Werten vom Typ `System.Data.ConnectionState`.

Die möglichen Werte dieser Enumeration finden Sie in Tabelle 9.2.

Tabelle 9.2 Werte für den ConnectionState

Enumerationswert	Bedeutung
`ConnectionState.Broken`	Die Verbindung ist unterbrochen – wird erst in zukünftigen Versionen unterstützt.
`ConnectionState.Close`	Die Verbindung ist geschlossen.
`ConnectionState.Connecting`	Die Verbindung wird gerade hergestellt – wird erst in zukünftigen Versionen unterstützt.
`ConnectionState.Executing`	Die Verbindung führt gerade einen Befehl aus – wird erst in zukünftigen Versionen unterstützt.
`ConnectionState.Fetching`	Die Verbindung ruft gerade Daten ab – wird erst in zukünftigen Versionen unterstützt.
`ConnectionState.Open`	Die Verbindung ist geöffnet.

Anstatt eine `Connection` mit `Close()` zu schließen, könnten Sie auch die `Dispose()`-Methode aufrufen. `Dispose()` führt intern auch einen `Close()` durch, hat aber den zusätzlichen Vorteil, dass die Ressource im Speicher sofort für den Garbage Collector zur Verfügung steht.

Das sind eigentlich schon die wichtigsten Sachen, die Sie über das `Connection`-Objekt wissen müssen. Lassen Sie mich aber noch auf zwei Themen eingehen, die für Sie sehr wichtig sein könnten.

Erstellen des ConnectionStrings

Der bislang in diesem Beispiel verwendete `ConnectionString` wird bei der Standardinstallation des Microsoft SQL Servers funktionieren, doch sehr oft werden auch zusätzliche Parameter benötigt oder ein anderer Authentifizierungstyp gewählt.

Zum Zusammensetzen eines gültigen `ConnectionStrings` gibt es auch ein spezielles Objekt, den `ConnectionStringBuilder`.

In Listing 9.2 will ich Ihnen eine kleine und ich denke auch selbst erklärende Funktion zeigen, wie Sie einen `SqlConnectionStringBuilder` nutzen, um einen `ConnectionString` dynamisch zu erstellen.

Listing 9.2 SqlConnection-StringBuilder

```
Private Function CreateConnectionString() As String

    Dim builder As New SqlConnectionStringBuilder
    builder.DataSource = "(local)\SqlExpress"
    builder.InitialCatalog = "Northwind"
    builder.IntegratedSecurity = True
    Return builder.ToString
End Function
```

Weitere Einstellungsmöglichkeiten können dabei sehr einfach über das Setzen von zusätzlichen Eigenschaften durchgeführt werden.

Connection Pooling

Wie bereits am Anfang dieses Kapitels erwähnt, unterstützen viele Datenbanksysteme Connection Pooling. Ein `Connection`-Objekt ist dabei tatsächlich nicht eine physikalische Verbindung zur Datenbank, sondern nur eine Ressource, die auf eine physikalische Verbindung zugreift. Der Pool wird dabei intern von ADO.NET verwaltet. Mit der Methode `Open()` wird die Ressource angefordert und mit `Close()` wieder freigegeben.

Standardmäßig ist Connection Pooling aktiviert, das heißt, Sie nutzen diese Technologie, ohne dass Sie es vielleicht wissen. Sie haben aber die Möglichkeit, Connection Pooling zu deaktivieren. Dazu müssen Sie lediglich den `ConnectionString` um `Pooling=False` erweitern. Seien Sie sich aber dessen bewusst, dass, wenn Sie Connection Pooling deaktivieren, jeder `Open()`-Aufruf tatsächlich einen zeitaufwändigen physikalischen Verbindungsaufbau darstellt.

Tipp

Probieren Sie, in einer Schleife Verbindungen zu öffnen und sofort wieder zu schließen. Messen Sie dabei die Zeit und probieren Sie es danach noch einmal mit deaktiviertem Connection Pooling. Es wird mit Sicherheit sehr viel länger dauern.

Alle `Connection`-Objekte mit einem identischen `ConnectionString` teilen sich dabei den Pool von physikalischen Connections auf. ADO.NET weist somit beim Aufruf von `Open()` eine freie physikalische Verbindung zu.

Die maximale und minimale Anzahl von gepoolten physikalischen Connections können Sie ebenso im `ConnectionString` mittels der Schlüsselwörter `MaxPoolSize` und `MinPoolSize` angeben.

9.4.2 SqlDataAdapter

Der `SqlDataAdapter` ist eines der zentralen Objekte innerhalb von ADO.NET. Der `SqlDataAdapter` kann Daten aus der Datenquelle holen und Daten auch wieder in die Datenbank zurückschreiben.

Der `SqlDataAdapter` besteht intern aus vier `Command`-Objekten:

- SelectCommand
- UpdateCommand
- InsertCommand
- DeleteCommand

Jedem dieser `Command`-Objekte muss eine `Connection` zugewiesen werden, die für die Ausführung des Befehls benutzt wird. Dabei kann die `Connection` bereits dem Konstruktor des `DataAdaper` übergeben werden. Der Konstruktor verlangt dann aber auch als ersten Parameter eine gültige Select-Anweisung oder auch bereits ein `SelectCommand`-Objekt.

Als gültiger `SelectCommandText` wird hierbei eine gültige SQL-Anweisung oder der Name einer StoredProcedure akzeptiert. Wenn Sie eine StoredProcedure

angeben, müssen Sie aber noch die Eigenschaft `CommandType` des `SelectCommand` auf den Wert `CommandType.StoredProcedure` setzen.

Eine Definition eines `DataAdapter` könnte wie folgt aussehen:

```
Private da As New SqlDataAdapter("Select * from Customers", con)
```

In diesem Beispiel werden die Daten aus der Tabelle *Customers* aus der Northwind-Datenbank selektiert.

Der Select-Befehl wird tatsächlich dann ausgeführt, wenn die `Fill()`-Methode des `DataAdapter`-Objekts ausgeführt wird. Der `DataAdapter` kümmert sich dabei selbst um den Verbindungsaufbau und -abbau. Sie müssen also nicht selbst die `Open()`-, `Close()`- oder `Dispose()`-Methode des `Connection`-Objekts aufrufen.

Die Methode `Fill()` schreibt dabei die Daten in eine `DataTable` oder in eine Tabelle innerhalb eines `DataSet`. Die entsprechende `DataTable` kann dann an beliebige Steuerelemente gebunden werden.

In einem neuen Windows-Anwendungsbeispiel habe ich zuerst, wie gerade beschrieben, den `SqlDataAdapter` definiert. Auf das Formular habe ich eine `DataGridView` gezogen und im `Form_Load()`-Ereignishandler des Formulars folgenden Programmcode hinterlegt:

```
Dim dt As New DataTable
da.Fill(dt)
DataGridView1.DataSource = dt
```

Hier wird zuerst ein Objekt vom Typ `DataTable` instanziiert, das anschließend vom `DataAdapter` befüllt wird. Danach wird die Tabelle noch als Datenquelle des Steuerelements angegeben und schon können Sie die Anwendung starten und Ihre Daten betrachten.

Eine Alternative dazu wäre die Befüllung einer Tabelle innerhalb eines `DataSet`. Der Code würde dann so aussehen:

```
Dim ds As New DataSet
da.Fill(ds, "Customers")
DataGridView1.DataSource = ds
DataGridView1.DataMember = "Customers"
```

Der Unterschied besteht darin, dass hier zuerst ein Objekt vom Typ `DataSet` instanziiert wird, und beim `Fill()` wird eine Tabelle innerhalb eines `DataSet` befüllt. Der Tabellenname, den ich hier zusätzlich angegeben habe, ist optional. Es empfiehlt sich aber, den Tabellennamen anzugeben, denn wenn mehrere Tabellen in einem `DataSet` vorhanden sind, ist der Code besser lesbar, wenn Sie über einen Namen auf die Tabellen zugreifen als über Indizes. Der Tabellenname muss dabei nicht zwingend dem Tabellennamen in der Datenbank entsprechen, würde aber die Lesbarkeit wiederum erhöhen.

Der Datenquelle des `DataGridView` wird hier das `DataSet` zugewiesen. Da ein `DataSet` aus mehreren Tabellen bestehen kann, muss zusätzlich die Eigenschaft `DataMember` des `DataGridView` mit dem Tabellennamen versorgt werden.

Beachten Sie bitte, dass Sie die erste Version mit der `DataTable` verwenden sollten, wenn Sie mit nur einer einzigen Tabelle arbeiten. Dieser Code ist performanter als das `DataSet`-Beispiel. Wenn aber mehrere Tabellen verarbeitet werden und diese Tabellen auch noch in Beziehung zueinander stehen sollen, dann verwenden Sie den `DataSet`-Ansatz.

In diesem Beispiel laden wir alle Daten aus einer beliebigen Tabelle. Das werden Sie normalerweise nicht tun, in der Regel wollen Sie nur eine bestimmte Auswahl von Daten anzeigen.

Info

Verzeihen Sie es mir, wenn ich in meinen Beispielen mit `Select *` auf die Tabellen zugreife. In der Realität sollten Sie tatsächlich nur die Spalten selektieren, die Sie auch sehen wollen.

Aber kommen wir nun zum Einschränken der angezeigten Daten. In der Tabelle *Customers* gibt es eine Spalte *Country*. Die möglichen Werte für die Spalte *Country* zeige ich in einer *ComboBox* auf dem Formular an und je nach Auswahl sollen die entsprechenden Daten angezeigt werden.

Den zugehörigen Programmcode sehen Sie in Listing 9.3.

Listing 9.3 Anzeige von Daten aufgrund Parameterauswahl

```
Imports System.Data.SqlClient
Public Class Form1

  Private con As New SqlConnection _
    (ConfigurationManager.ConnectionStrings _
      ("NorthwindConnection").ConnectionString)

  Private da As New SqlDataAdapter _
    ("Select * from Customers where Country=@country", con)

  Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load

    da.SelectCommand.Parameters.AddWithValue _
            ("@country", ComboBox1.Text)

    Dim daCountry As New SqlDataAdapter(
      "Select distinct country from Customers", con)

    Dim dt As New DataTable
    daCountry.Fill(dt)
    ComboBox1.DataSource = dt
    ComboBox1.DisplayMember = "Country"

  End Sub

  Private Sub ComboBox1_SelectedIndexChanged _
     (ByVal sender As System.Object, _
      ByVal e As System.EventArgs) _
      Handles ComboBox1.SelectedIndexChanged

      da.SelectCommand.Parameters("@country"). _
         Value = ComboBox1.Text
```

```
        Dim ds As New DataSet
        da.Fill(ds, "Customers")
        DataGridView1.DataSource = ds
        DataGridView1.DataMember = "Customers"
    End Sub
End Class
```

Listing 9.3 (Forts.) Anzeige von Daten aufgrund Parameterauswahl

An dieser Stelle habe ich im Select-Befehl einen Parameter angegeben. Dadurch kann man sehr bequem Einschränkungen definieren. Anschließend habe ich diesen Parameter der Parameterauflistung des `SelectCommand` hinzugefügt. Jedes Mal, wenn die Auswahl für das Land geändert wird, wird der Parameter neu befüllt und der Select-Befehl mit diesem Wert ausgeführt.

Außerdem wird im `Form_Load()`-Ereignishandler noch die ComboBox mit allen in der Tabelle *Customers* enthaltenen Ländern befüllt. Durch die Angabe des Schlüsselworts `Distinct` wird jedes Land auch nur ein Mal in der `ComboBox` angezeigt.

Das Ergebnis sehen Sie in Abbildung 9.17.

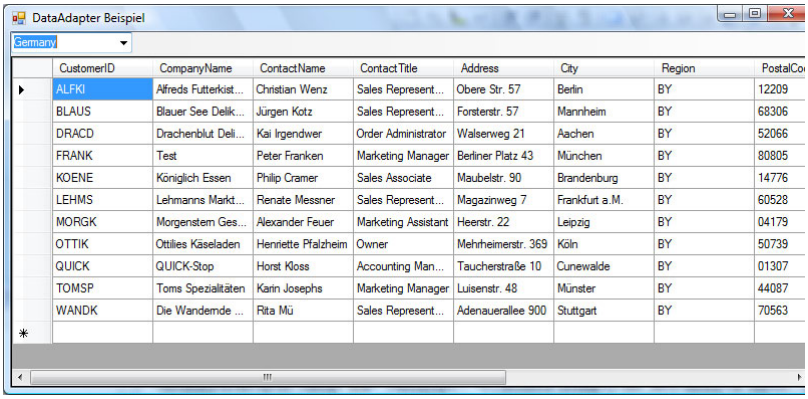


Abbildung 9.17 Selektion der Daten durch einen Parameter

Bislang haben wir nur die Präsentation von Daten mittels eines `DataAdapter` besprochen, kommen wir nun dazu, wie wir Änderungen von Daten zurückschreiben können.

Für die Anzeige wurde das `SelectCommand` ausgeführt. Die restlichen drei internen Command-Objekte werden beim Zurückschreiben der Daten benötigt. Diese Funktionalität wird durch den Aufruf der `Update()`-Methode ausgeführt.

Beim Aufruf der `Update()`-Methode brauchen Sie sich, genauso wie beim Aufruf des `Fill()`-Befehls, nicht um den Verbindungsaufbau und -abbau zu kümmern.

Der `Update()`-Methode muss dabei ein Array von `DataRows`, ein `DataSet` oder eine `DataTable` übergeben werden. Die `Update()`-Methode parst dann durch die übergebene Datenmenge und sucht dabei nach neuen, gelöschten oder geänderten Datensätzen – diese Information besitzen die Datensätze selbst – und sendet die entsprechenden Update-, Delete- und Insert-Anweisungen an die Datenbank.

Den SelectCommandText haben wir im Konstruktor des DataAdapter angegeben. Die Befehlstexte für das UpdateCommad, InsertCommand und DeleteCommand müssen Sie selbst setzen. Sie können dies durch das Setzen der CommandText-Eigenschaft für die einzelnen Commands mit der gewünschten SQL-Anweisung machen. Dies funktioniert genauso wie bei einem normalen Command-Objekt. Ein Beispiel dazu implementiere ich im nächsten Abschnitt **9.4.3**.

Auch hier ist es möglich, als CommandText den Namen einer StoredProcedure anzugeben. Vergessen Sie dann aber nicht, auch den CommandType auf den Wert CommandType.StoredProcedure zu setzen.

Es gibt aber auch noch die Möglichkeit, sich die entsprechenden Aktualisierungskommandos automatisch generieren zu lassen. Dies ist in manchen Fällen durchaus sinnvoll, denn damit kann man sich eine Menge Arbeit sparen. Seien Sie sich aber dessen bewusst, dass diese generisch generierten SQL-Anweisungen, aufgrund ihrer Flexibilität, nicht unbedingt immer performant arbeiten werden.

Zum Generieren der Commands können Sie ein CommandBuilder-Objekt verwenden, wie Sie hier sehen.

```
Private Sub GenerateCommandText (ByVal da As SqlDataAdapter)
    Dim builder As New SqlCommandBuilder(da)
    da.UpdateCommand = builder.GetUpdateCommand()
    da.InsertCommand = builder.GetInsertCommand()
    da.DeleteCommand = builder.GetDeleteCommand()
End Sub
```

Dieser Routine wird ein SqlDataAdapter-Objekt übergeben, das dem Konstruktor des SqlCommandBuilder übergeben wird. Aufgrund des SelectCommand, das dem SqlDataAdapter zugewiesen wird, kann jetzt der SqlCommandBuilder die entsprechenden Command-Objekte erzeugen.

9.4.3 SqlCommand

Das nächste Objekt, das wir betrachten wollen, ist das SqlCommand-Objekt. Mit diesem Objekt können beliebige Befehle an die Datenbank gesendet werden. Beim SqlCommand sind Sie aber für die Verbindung zur Datenquelle selbst verantwortlich. Das heißt, Sie müssen die entsprechenden Open()- und Close()-Methoden des zugehörigen Connection-Objekts selbst aufrufen.

Dem Konstruktor eines SqlCommand-Objekts können bis zu drei unterschiedliche Parameter übergeben werden.

- CommandText

  Der Befehlstext, der an die Datenquelle gesendet wird. Kann eine SQL-Anweisung oder der Name einer StoredProcedure sein.

- Connection

  Die Connection, an die der Befehl gesendet wird

- Transaction

  Falls der Befehl an einer Transaktion teilnimmt, kann die entsprechende Transaktion hier angegeben werden.

Die entsprechenden Befehle können dann mit einem `Execute`-Befehl ausgeführt werden. Das `Command`-Objekt stellt dabei drei unterschiedliche Execute-Variationen zur Verfügung:

- ExecuteNonQuery()

  Führt einen Befehl gegen eine Datenbank aus und bringt kein Ergebnis in Form einer Abfrage zurück. Diesen Execute verwenden Sie beim Senden eines Update-, Delete- oder Insert-Befehls. Der Rückgabewert dieser Methode ist ein Integer-Wert, der die Anzahl der Datensätze angibt, die von der Anweisung betroffen waren.

- ExecuteReader()

  Führt einen Befehl gegen eine Datenbank aus und gibt eine Query an einen DataReader zurück. Diesen Execute verwenden Sie beim Absetzen eines Select-Befehls.

- ExecuteScalar()

  Führt einen Befehl gegen eine Datenbank aus und gibt einen eindeutigen Wert vom Typ Object zurück, der sofort in den gewünschten Typ umgewandelt werden kann, ohne zuerst Daten in einen DataReader oder eine DataTable zu laden. Es kann aber nur ein einziger, skalarer Wert zurückgegeben werden. Diesen Execute verwenden Sie beim Absetzen eines Select-Befehls, der ein eindeutiges Ergebnis zurückbringt, wie zum Beispiel *Select Count(*) from tablename*.

Nur für das `SqlCommand` gibt es auch noch die Methode `ExecuteXmlReader()` zum Auslesen von XML-Streams aus dem SQL Server. Der Rückgabewert dieser Methode ist ein `XmlReader`-Objekt.

Für die Execute-Methoden des SqlCommand-Objekts gibt es auch die Möglichkeit, die Befehle asynchron abzusetzen. Dazu stehen Ihnen jeweils die Methoden `BeginExecuteReader()`, `BeginExecuteNonQuery()` und `BeginXmlReader()` zum Starten des asynchronen Aufrufs zur Verfügung. Mit `EndExecuteNonQuery()`, `EndExecuteReader()` und `EndXmlReader()` beenden Sie den asynchronen Aufruf. Diese Methoden sind nur für den Microsoft SQL Server verfügbar.

Im Folgenden will ich ein paar kleine Codebeispiele für das `SqlCommand`-Objekt zeigen.

ExecuteReader

Im ersten Schritt will ich mit einem `DataReader` eine `ComboBox` befüllen. Dafür verwende ich die `ExecuteReader()`-Methode des `Command`-Objekts.

Als `Connection` verwende ich dazu dasselbe `Connection`-Objekt, das ich bereits bei den vorigen Codebeispielen benutzt habe.

Listing 9.4 Command-Object ExecuteReader()

```
Imports System.Data.SqlClient
Imports System.Configuration

Public Class Form1

  Private con As New SqlConnection _
    (ConfigurationManager.ConnectionStrings _
     ("NorthwindConnection").ConnectionString)

  Private cmd As New SqlCommand

  Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load

    cmd.Connection = con
    cmd.CommandText = "Select distinct country from customers"
    con.Open()

    Dim dr As SqlDataReader = cmd.ExecuteReader()
    Do While dr.Read
      ComboBox1.Items.Add(dr("Country"))
    Loop
    dr.Close()
    con.Close()
    If ComboBox1.Items.Count > 0 Then
       ComboBox1.SelectedIndex = 0
    End If
  End Sub
End Class
```

In Listing 9.4 sehen wir, wie die `Connection`-Eigenschaft und der `CommandText` des `Command`-Objekts gesetzt werden. Danach wird die `ExecuteReader()`-Methode aufgerufen und der erzeugte Reader durchlaufen. Für jede Zeile aus dem Reader wird dabei ein Eintrag in die `ComboBox` hinzugefügt. Anschließend werden der `DataReader` und die `Connection` geschlossen. Der Reader braucht zum Durchlaufen ein geöffnetes `Connection`-Objekt, weil er im Grunde ein Cursor auf der Datenbank ist. Diese `Connection` muss dem `DataReader` dabei auch exklusiv zur Verfügung stehen.

ExecuteScalar

Im zweiten Schritt will ich wissen, wie viele Kunden aus einem bestimmten Land kommen. Dazu verwende ich die `ExecuteScalar()`-Methode mit einem Parameter für das ausgewählte Land.

Dafür habe ich eine neue Schaltfläche auf das Formular gezogen und den Code aus Listing 9.5 hinterlegt.

Listing 9.5 Command-Object mit ExecuteScalar()

```
Private Sub btnExecuteScalar_Click
  (ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnExecuteScalar.Click

  cmd = New SqlCommand
  cmd.Connection = con
  cmd.CommandText = "Select Count(*) from Customers " & _
     "where country = @country"
```

```
  cmd.Parameters.AddWithValue("@country", ComboBox1.Text)
  con.Open()
  Dim Anzahl As Integer = CType(cmd.ExecuteScalar, Integer)
  con.Close()
  MessageBox.Show("Es sind " & _
    Anzahl.ToString & " Kunden aus diesem Land")
End Sub
```

Listing 9.5 (Forts.) Command-Object mit ExecuteScalar()

Im `CommandText` habe ich einen Parameter angegeben, den ich der Parameterauflistung des `Command`-Objekts hinzufüge und auf den entsprechenden Wert setze. Beim Aufruf der `ExecuteScalar()`-Methode weise ich den gewünschten Wert sofort einer Variablen dazu.

Die `ExecuteScalar()`-Methode ist dabei wieder mit dem Öffnen und Schließen des Verbindungsobjekts umschlossen.

So, jetzt wird es aber auch Zeit, ein paar Worte über die Parameterobjekte zu verlieren.

Parameter

Im `CommandText` in Listing 9.5 habe ich in der *Where*-Bedingung einen Platzhalter gesetzt. An dieser Stelle muss danach ein Parameter mit dem entsprechenden Wert übergeben werden. Und für diese Zwecke können Sie danach `Parameter`-Objekte an die `Parameters`-Auflistung von `Command`-Objekten anhängen und dem Parameter einen entsprechenden Wert zuweisen.

Stattdessen hätte ich aber genauso den `CommandText` dynamisch zusammensetzen können, in etwa so:

```
"Select Count(*) from Customers " & _
    "where Country = '" & ComboBox1.Text & "'"
```

Wo liegen aber jetzt die Vorteile der Parameterobjekte gegenüber der dynamischen String-Zusammensetzung?

Zum einen wird Ihr Code bei weitem übersichtlicher, denn bedenken Sie, dass bei mehreren Parametern die Strings immer wieder zusammengesetzt werden müssen. Außerdem müssen Sie dann noch einige Regeln beachten, zum Beispiel, dass ein String in einfachen Hochkommata übergeben werden muss, in dem String selbst jedoch kein einfaches Hochkomma stehen darf, dass Zahlen keine Kommata enthalten dürfen, sondern einen Punkt als Dezimaltrennzeichen usw.

Wenn Sie alle diese Regeln beachten, dann wird Ihr Code bald nicht mehr lesbar, und je mehr Variablen Ihre SQL-Anweisung enthält, desto länger wird die Fehlersuche brauchen.

Ein weiterer und noch viel wichtigerer Punkt ist die Sicherheit. Variablenwerte können sehr oft auch aus frei editierbaren Textfeldern kommen, das bedeutet, dass der Benutzer beliebigen Text eingeben kann. Das könnte wiederum eine SQL-Anweisung sein, dies nennt man SQL-Injection. Bei Parameterobjekten kann der Befehlstext nicht manipuliert werden, was bei einem dynamisch erzeugten String leider nicht der Fall ist.

Achtung

Das @-Zeichen vor dem eigentlichen Parameternamen ist für den SQL-Server das Zeichen für eine Variable. Bei anderen Datenbanksystemen werden hier andere Zeichen erwartet, bei Oracle zum Beispiel ein Doppelpunkt (:Country) oder bei Access nur ein Fragezeichen ohne Parametername (?).

Parameter können dabei zu einem `Command`-Objekt mit der Methode `Add()` oder `AddWithValue()` hinzugefügt werden. Bei `Add()` kann neben dem Parameternamen auch noch der entsprechende Datentyp angegeben werden. Die Methode `AddWithValue()` hingegen erwartet den Namen des Parameters und den entsprechenden Wert.

Auf einen Parameter kann man später jederzeit auch über die Parameteraufllistung des Objekts wieder zugreifen.

So, dann fehlt uns nur noch die `ExecuteNonQuery()`-Methode.

ExecuteNonQuery

Im letzten Schritt will ich Datenänderungen mit `ExecuteNonQuery()` vornehmen. Dabei ändere ich den Wert der Spalte `Region` auf den Wert »*BY*« und mache anschließend die Änderung wieder rückgängig, damit ich das Beispiel auch noch öfter demonstrieren kann. Ich füge also eine neue Schaltfläche auf mein Formular und den Programmcode aus Listing 9.6 hinzu.

Listing 9.6 Command-Object mit ExecuteNonQuery()

```
Private Sub btnExecuteNonQuery_Click _
  (ByVal sender As System.Object, ByVal e As EventArgs) _
   Handles btnExecuteNonQuery.Click

  cmd = New SqlCommand
  cmd.Connection = con
  cmd.CommandText = "Update Customers " & _
    "set Region = 'BY' where Country = @Country"
  cmd.Parameters.AddWithValue("@Country", ComboBox1.Text)
  con.Open()
  Dim betroffeneSaetze As Integer = cmd.ExecuteNonQuery
  cmd.CommandText = "Update Customers " & _
    "set Region = '' where Country = @Country"
    con.Close()
    MessageBox.Show("Es sind " & betroffeneSaetze.ToString & _
      " Sätze geändert worden")
End Sub
```

Auch wenn die Methode `ExecuteNonQuery()` eigentlich kein Abfrageergebnis zurückgibt, so hat Sie, wie wir gesehen haben, trotzdem einen Rückgabewert, nämlich die Anzahl der betroffenen Sätze.

Was mich jedoch am oberen Beispiel ein bisschen stört, ist die Tatsache, dass ich mich darauf verlasse, dass beide `ExecuteNonQuery()`-Methoden fehlerfrei durchlaufen werden. Es kann immer wieder Situationen geben, dass beim Ausführen des zweiten Befehls ein Laufzeitfehler auftritt, aus welchen Gründen auch immer.

Ich möchte jedoch die Anforderung erfüllen, dass der erste Befehl rückgängig gemacht wird, falls der zweite Befehl schief geht, also alles oder nichts. Und genau für solche Zwecke gibt es Transaktionen.

Transaktionen

Bevor ich zeige, wie Sie die `Command`-Objekte innerhalb einer Transaktion ausführen lassen, will ich ganz kurz ein paar grundsätzliche Sachen zu Transaktionen beschreiben.

Transaktionen sind eine Folge von Datenbankoperationen, die als eine Einheit ausgeführt werden sollen. Sie werden eingesetzt, um bei mehreren Arbeitsschritten die Integrität der Daten zu gewährleisten. Für Transaktionen gilt das ACID-Prinzip, das in Tabelle 9.3 erläutert wird.

Tabelle 9.3 Transaktionen – das ACID-Prinzip

Prinzip	Beschreibung
A – Atomicity	Eine Transaktion erlaubt die Gruppierung von mehreren Kommandos in eine unteilbare, quasi atomare Aktion. Das bedeutet, entweder es sind alle Teile der Transaktion erfolgreich oder keiner. Schlägt ein Teil der Transaktion fehl, werden alle bis dahin vorgenommenen Änderungen rückgängig gemacht.
C – Consistency	Transaktionen operieren immer auf konsistenten Ansichten des Datenbestands. Wenn sie beginnen und enden, befinden sich alle Daten in einem konsistenten Zustand. Beispielsweise befindet sich eine Datenbank in einem inkonsistenten Zustand, wenn zwei Datensätze denselben Primärschlüssel haben oder ein Fremdschlüsselsatz keinen Primärschlüsselsatz referenziert. Während einer Transaktion darf die Konsistenz verletzt werden, allerdings darf keine andere Transaktion die Zwischenwerte sehen.
I – Isolation	Eine Transaktion muss immer so ablaufen, als wäre sie allein in der Datenbank. Die Änderungen anderer Transaktionen müssen unsichtbar für sie sein.
D – Durability	Wenn eine Transaktion erfolgreich beendet ist, muss die Persistenz der Änderungen garantiert sein. Analog dazu muss gewährleistet sein, dass vor dem Transaktionsende KEINE Änderungen der Transaktion bei einem Systemausfall gespeichert werden.

Transaktionen werden von den Datenbanksystemen mit Hilfe von Datensatzsperren implementiert. Änderungen im Zuge einer Transaktion notiert die Datenbank in einem Puffer.

Zur Steuerung einer Transaktion gibt es drei grundlegende Kommandos: `Begin()`, `Commit()` und `Rollback()`. Das `Begin()`-Statement bestimmt den Beginn einer Transaktion. Alle folgenden Aktionen laufen im Kontext dieser Transaktion. Das `Commit()`-Statement schließt eine Transaktion ab, `Rollback()` nimmt alle Änderungen seit `Begin()` zurück.

Diese Kommandos werden von den meisten Datenbanksystemen in deren SQL-Dialekt implementiert.

In ADO.NET werden Transaktionen über das `Connection`-Objekt gestartet. Jedes an der Transaktion teilnehmende `Command`-Objekt muss dieser Transaktion zugewiesen werden. Zum Starten einer Transaktion wird die `BeginTransaction()`-Methode ausgeführt. Diese Methode liefert ein Transaktionsobjekt zurück, das für die weitere Verwaltung der Transaktion zuständig ist. Mit der `Commit()`-Methode dieses Objekts wird die Transaktion abgeschlossen, `Rollback()` setzt die Transaktion zurück.

In unserem kleinen Beispiel will ich jetzt die beiden `ExecuteNonQuery()`-Methoden in einer Transaktion zusammenfassen. Dazu lege ich einen weiteren Button auf mein Formular, unter dem ich den Programmcode aus Listing 9.6 erst kopiere, um ihn dann an ein transaktionelles Verhalten anzupassen. Das Ergebnis sehen Sie in Listing 9.7.

Listing 9.7 Command-Objekte mit Transaktionen

```
Private Sub btnTransaktion_Click(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles btnTransaktion.Click

  Dim transStarted As Boolean
  cmd = New SqlCommand
  cmd.Connection = con
  cmd.CommandText = "Update Customers " & _
    "set Region = 'BY' where Country = @Country"
  cmd.Parameters.AddWithValue("@Country", ComboBox1.Text)
  Dim tx As SqlTransaction = Nothing
  Try
    con.Open()
    tx = con.BeginTransaction
    transStarted = True
    cmd.Transaction = tx
    Dim betroffeneSaetze As Integer = cmd.ExecuteNonQuery
    cmd.CommandText = "Update Customers " & _
      "set Region = '' where Country = @Country"
    tx.Commit()
    MessageBox.Show("Es sind " & betroffeneSaetze.ToString & _
      " Sätze geändert worden")
  Catch ex As Exception
    If transStarted Then
      tx.Rollback()
    End If
    MessageBox.Show(ex.Message)
  Finally
    If con.State = ConnectionState.Open Then
      con.Close()
    End If
  End Try
End Sub
```

In diesem geänderten Beispiel habe ich als Erstes eine Fehlerbehandlung eingebaut. Läuft die Transaktion erfolgreich, so wird sie als gültig erklärt und bestätigt. Im Fehlerfall führe ich einen Rollback durch, also werden alle Änderungen zurückgenommen, jedoch nur für den Fall, dass die Transaktion bereits gestartet wurde. Das Transaktionsobjekt wird vor dem ersten Ausführen eines Befehls dem `Command`-Objekt über dessen Eigenschaft `Transaction` zugewiesen.

Tipp

Eine Alternative, vor allem wenn Transaktionen über mehrere Datenbankverbindungen hinweg gehen sollen, stellen die Klassen im Namespace `System.Transactions` dar. Dazu müssen Sie aber erst eine Referenz auf die Bibliothek System.Transactions.dll zu Ihrem Projekt hinzufügen.

Asynchrone Aufrufe im SQL Server

Zum Abschluss will ich Ihnen noch ein kleines Beispiel zeigen, wie Sie einen Befehl asynchron aufrufen können.

Dies ist jedoch nur, wie bereits eingangs erwähnt, mit dem `SqlCommand`-Objekt möglich.

Bevor wir den Programmcode betrachten, müssen wir noch eine kleine Vorkehrung treffen. Damit wir asynchrone Aufrufe an die Datenbank absetzen können, müssen wir der Datenbank auch mitteilen, dass wir asynchrone Aufrufe durchführen wollen. Das können wir machen, indem wir den String `Asynchronous Processing=True` dem `ConnectionString` hinzufügen.

Listing 9.8
Asynchroner Aufruf des Connection-Object

```
Private Sub btnAsynchron_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnAsynchron.Click

  Dim builder As New SqlConnectionStringBuilder
  builder.ConnectionString = con.ConnectionString
  builder.AsynchronousProcessing = True
  Dim asyncCon As New SqlConnection(builder.ToString)

  cmd = New SqlCommand
  cmd.Connection = asyncCon
  cmd.CommandText = "Update Customers set Region = 'BY'"

  asyncCon.Open()
  Dim result As IAsyncResult = cmd.BeginExecuteNonQuery
  While Not result.IsCompleted
    Dim waits As Integer
    TextBox1.Text = waits.ToString
    waits += 1
  End While
  Dim betroffeneSaetze As Integer = _
    cmd.EndExecuteNonQuery(result)
  cmd.CommandText = "Update Customers set Region = ''"
  cmd.ExecuteNonQuery()
  asyncCon.Close()
  MessageBox.Show("Es sind " & betroffeneSaetze.ToString & _
    " Sätze geändert worden")
End Sub
```

Der asynchrone Aufruf ist in Listing 9.8 abgebildet.

Zuerst kümmern wir uns dabei um den `ConnectionString`. Ich instanziiere dabei ein Objekt vom Typ `SqlConnectionStringBuilder`, den ich mit dem ursprünglichen `ConnectionString` initialisiere. Dann füge ich dem `SqlConnectionStringBuilder` die Möglichkeit hinzu, asynchrone Prozesse aufzurufen. Mit diesem neuen `ConnectionString` instanziiere ich anschließend ein neues `Connection`-Objekt, über das ich den asynchronen Aufruf durchführe.

Die Methode `BeginExecuteNonQuery()` hat einen Rückgabewert `result` vom Typ `IAsyncResult`.

Nachdem der Befehl asynchron abgesetzt wurde, lasse ich eine Schleife so lange laufen, bis der Befehl abgearbeitet wurde. Innerhalb der Schleife erhöhe ich einen Zähler in einer `TextBox`.

Nachdem die Eigenschaft `IsCompleted` des `result`-Objekts `True` geworden ist, rufe ich mittels `EndExecuteNonQuery()` den Rückgabewert der Methode auf. `EndExceuteNonQuery()` braucht als Parameter noch das `result`-Objekt, das der Rückgabewert der `BeginExecuteNonQuery()`-Methode war. Da die Anweisung für das Update nicht sehr lange braucht, wird auch die Zahl, die in der `TextBox` dargestellt wird, sehr niedrig sein. Bei länger laufenden Aktionen kann das aber sehr viel anders aussehen.

Somit wären wir dann fürs Erste mit den providerabhängigen Objekten durch und wechseln zur Beschreibung der providerunabhängigen Objekte.

9.4.4 DataSet

Das zentrale providerunabhängige Objekt ist das `DataSet`. Ich habe das `DataSet` bereits bei der Erläuterung des `DataAdapter` verwendet, will aber nun noch etwas genauer darauf eingehen.

Ein `DataSet` ist prinzipiell eine Auflistung von Datentabellen, die durch Relationen miteinander in Beziehung stehen können.

Im ersten Schritt will ich eine Tabelle in einem `DataSet` über einen `DataAdapter` befüllen. Ich verwende dazu wieder die *Customers*-Tabelle aus den vorherigen Beispielen, lade jedoch nur Kunden, deren *CompanyName* mit dem Buchstaben A beginnt. Die Daten hänge ich dann wieder an eine `DataGridView`. Den Programmcode sehen Sie in Listing 9.9.

Listing 9.9 Befüllen eines DataSet

```
Imports System.Data.SqlClient
Imports System.Configuration

Public Class Form1

  Private con As New SqlConnection _
    (ConfigurationManager.ConnectionStrings
    ("NorthwindConnection").ConnectionString)

  Private ds As New DataSet

  Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load

    Dim da As New SqlDataAdapter _
      ("Select * from Customers where CompanyName like 'A%'", _
        con)
    da.Fill(ds, "Customers")
    DataGridView1.DataSource = ds
    DataGridView1.DataMember = "Customers"

  End Sub
End Class
```

So, nachdem wir jetzt ein paar Daten geladen haben, können wir uns einige wichtige Eigenschaften und Methoden des `DataSet` anschauen.

XML-Serialisierung

Eine der großen Stärken eines `DataSet` ist seine automatische Serialisierung zu XML. Das bedeutet, wir können jetzt mit einer einzigen Zeile Programmcode den Inhalt des `DataSet` zu XML serialisieren.

Nachdem wir folgende Zeile noch zu unserem Miniprogramm hinzufügen, finden wir nach dem nächsten Programmstart die Datei *customers.xml* in unserem Applikationsverzeichnis.

```
ds.WriteXml(Application.StartupPath & "\customers.xml")
```

Die Datei beinhaltet alle Daten im XML-Format, wie wir in der folgenden Abbildung sehen.

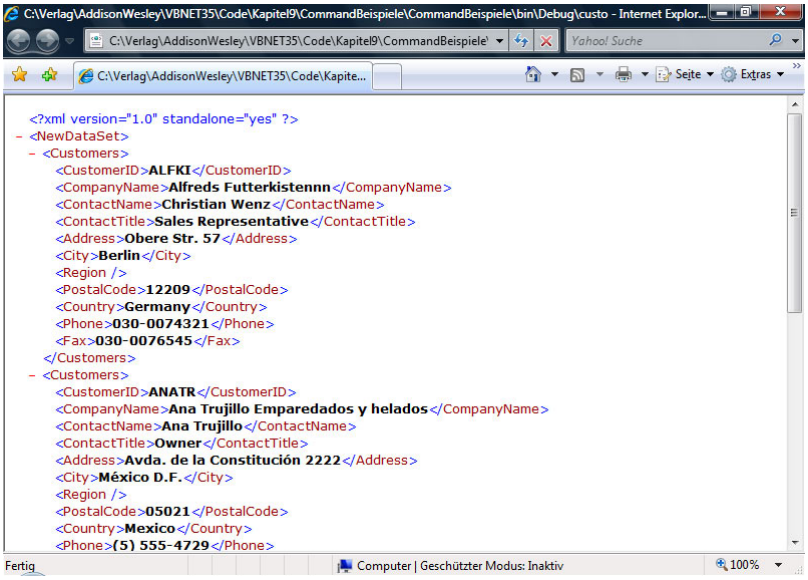


Abbildung 9.18
customers.xml

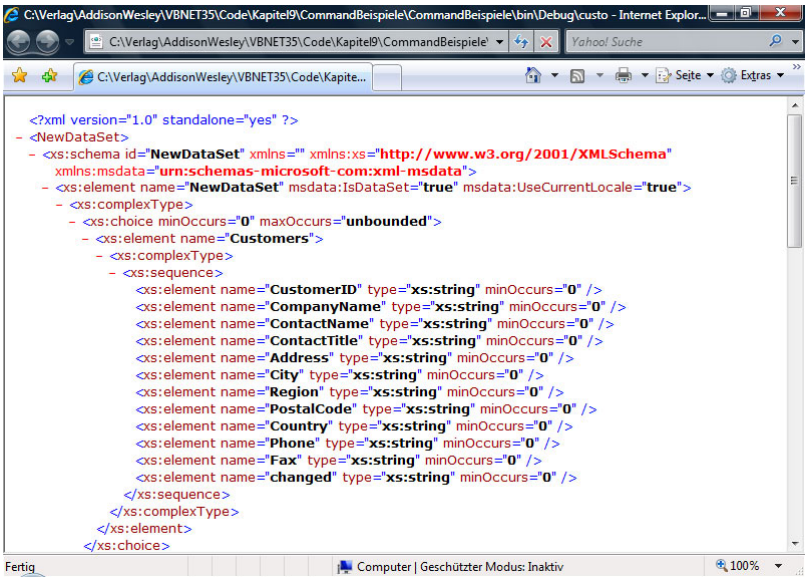
Oftmals ist es jedoch sinnvoll, auch die Schema-Information unserer Daten mit abzuspeichern.

Das können wir erreichen, indem wir die `WriteXml()`-Methode mit einem weiteren Parameter aufrufen.

```
ds.WriteXml(Application.StartupPath & _
   "\customers.xml", XmlWriteMode.WriteSchema)
```

Die geänderte *customers.xml* sehen Sie in Abbildung 9.19.

Abbildung 9.19
customers.xml mit Schema-Informationen



Außer der `WriteXml()`-Methode bietet das `DataSet` noch weitere Funktionen im Zusammenhang mit XML:

- GetXml()

  Gibt die XML-Darstellung des DataSet als String zurück.

- GetXmlSchema()

  Gibt das XML-Schema des DataSet als String zurück.

- ReadXml()

  Liest eine XML-Datei ein und wandelt die XML-Information in ein DataSet, somit können mit WriteXml() geschriebene Daten später wieder eingelesen werden.

- ReadXmlSchema()

  Liest XML-Schema-Informationen ein, um somit das Schema eines DataSet festzulegen.

- WriteXmlSchema()

  Schreibt nur die Schema-Informationen eines DataSet in eine XML-Struktur.

In den ersten .NET-Versionen hat sich ein `DataSet` bei der Übertragung innerhalb eines Netzwerks automatisch zu XML serialisiert und auf der Gegenseite wieder deserialisiert. Damit wollte man Probleme mit Firewalls umgehen. Leider wurden die zu übertragenden Datenmengen durch die XML-Serialisierung aufgebläht. Wollte man die Daten binär übertragen, hatte man in der Vergangenheit keine Chance dazu. In ADO.NET 2.0 wurde eine neue Eigenschaft `RemotingFormat` eingeführt. Der Standardwert für das `RemotingFormat` ist dabei `SerializationFormat.Xml`. Sie können aber, falls Sie eine binäre Serialisierung wünschen, den Wert auf `SerializationFormat.Binary` setzen.

Weitere wichtige Eigenschaften und Methoden

In der Tabelle 9.4 sehen Sie eine Übersicht über weitere wichtige Methoden und Eigenschaften eines `DataSet`.

Tabelle 9.4 Weitere wichtige Methoden/Eigenschaften eines DataSet

Methode/Eigenschaft	Beschreibung
`AcceptChanges()`	Nimmt alle Änderungen seit dem letzten Laden an, Daten können nicht mehr zurückgenommen werden (Commit).
`RejectChanges()`	Nimmt alle Änderungen seit dem letzten Laden oder seit dem Aufruf der `AcceptChanges()`-Methode zurück (Rollback).
`Clear()`	Löscht alle Daten im `DataSet`, nicht aber die Tabellendefinitionen.
`Clone()`	Kopiert alle Tabellendefinitionen in ein neues `DataSet`, nicht aber die Daten.
`Copy()`	Kopiert alle Tabellendefinitionen und Daten in ein neues `DataSet`.
`CreateDataReader()`	Erzeugt ein Array von `DataReader` (für jede `DataTable` ein `DataReader`).
`GetChanges()`	Erzeugt ein neues `DataSet`, in dem alle geänderten Datensätze des originalen `DataSet` stehen.
`HasChanges`	Zeigt an, ob in einem `DataSet` Daten geändert wurden.
`Relations`	Gibt die Auflistung aller Relationen für dieses `DataSet` zurück.
`Tables`	Gibt die Auflistung aller Tabellen für dieses `DataSet` zurück.

Als Nächstes wollen wir uns um das `DataTable`-Objekt kümmern.

DataTable

Eine `DataTable` enthält die Daten, die eine Select-Anweisung aus einer Datenquelle liefert. Intern ist eine `DataTable` eine Auflistung von Spalten und Zeilen.

Im nachfolgenden Beispielcode wollen wir auf die `DataTable` zugreifen, die wir im vorigen `DataSet`-Beispiel befüllt haben, und die Zeilen- und Spaltenauflistung durchlaufen.

Listing 9.10 Durchlaufen der Columns- und Rows-Auflistung einer DataTable

```
Private Sub btndatatable_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btndatatable.Click

  Dim dt As DataTable = ds.Tables("Customers")
  For Each dc As DataColumn In dt.Columns
    Debug.WriteLine(dc.ColumnName & " " & dc.DataType.ToString)
  Next
  For Each dr As DataRow In dt.Rows
    Debug.Write(dr("CustomerId").ToString & " ")
  Next
End Sub
```

In Listing 9.10 initialisieren wir ein `DataTable`-Objekt, indem wir es auf die *Customers*-Tabelle in unserem `DataSet` referenzieren. Anschließend durchlaufen wir in einer `For Each`-Schleife die Spaltenauflistung der Tabelle und geben den Spaltennamen und dessen Datentyp aus. Am Ende der Routine durchlaufen wir in einer weiteren `For Each`-Schleife sämtliche Zeilen und geben dabei die *CustomerId*-Spalte aus.

Die Debug-Ausgabe sehen Sie in Abbildung 9.20.

Abbildung 9.20
Debug-Ausgabe

In der Tabelle 9.5 ist eine Auflistung der wichtigsten Eigenschaften und Methoden einer `DataTable` enthalten. Sie werden dabei sehr viele Überschneidungen zum `DataSet` feststellen.

Tabelle 9.5
Wichtige Methoden/Eigenschaften einer DataTable

Methode/Eigenschaft	Beschreibung
`AcceptChanges()`	Nimmt alle Änderungen seit dem letzten Laden an, Daten können nicht mehr zurückgenommen werden (Commit).
`RejectChanges()`	Nimmt alle Änderungen seit dem letzten Laden oder seit dem Aufruf der `AcceptChanges()`-Methode zurück (Rollback).
`Clear()`	Löscht alle Daten in der `DataTable`, nicht aber die Tabellendefinitionen.
`Clone()`	Kopiert die Tabellendefinitionen, nicht aber die Daten.
`Columns`	Gibt die Auflistung aller Spalten der Tabelle zurück.
`Copy()`	Kopiert alle Tabellendefinitionen und Daten in eine neue `DataTable`.
`CreateDataReader()`	Erzeugt einen `DataReader`.
`DataSet`	Gibt das `DataSet` zurück, zu dem die Tabelle gehört.

Methode/Eigenschaft	Beschreibung
`GetChanges()`	Erzeugt eine neue `DataTable`, in der alle geänderten Datensätze der originalen Tabelle stehen.
`NewRow()`	Erzeugt eine neue `DataRow`, passend zum Tabellenschema.
`ReadXml()`	Liest eine XML-Datei ein und wandelt die XML-Information in eine `DataTable`. Somit können mit `WriteXml()` geschriebene Daten später wieder eingelesen werden.
`ReadXmlSchema()`	Liest XML-Schema-Informationen ein, um somit das Schema einer `DataTable` festzulegen.
`Rows`	Gibt die Auflistung aller Zeilen innerhalb der `DataTable` zurück.
`TableName`	Gibt den Tabellennamen der Tabelle zurück.
`WriteXml()`	Schreibt die Daten einer `DataTable` in eine XML-Struktur.
`WriteXmlSchema()`	Schreibt die Schema-Informationen einer `DataTable` in eine XML-Struktur.

Tabelle 9.5 (Forts.) Wichtige Methoden/Eigenschaften einer DataTable

DataView

Eine `DataView` ist eine Sicht auf eine Tabelle. Mittels einer `DataView` können Sie Datensätze sortieren und filtern. Dabei gibt es zwei unterschiedliche Filter: einen `RowFilter`, der einen Filter aufgrund der Datenbestände setzt, und einen `RowStateFilter`, der einen Filter aufgrund des Zustands einer Datenzeile (neue Zeile, geänderte Zeile, gelöschte Zeile) setzt.

Im folgenden Beispiel setzen wir einen Filter aufgrund eines ausgewählten Lands, den wir in einer `ComboBox` anbieten. Die `DataSource`-Eigenschaft des `DataGridView` wird schließlich auf die `DataView` gesetzt, wie Sie in Listing 9.11 sehen.

Listing 9.11 DataView

```
Private Sub btnDataView_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnDataView.Click

  Dim dv As DataView = ds.Tables("Customers").DefaultView
  dv.RowFilter = "Country = '" & ComboBox1.Text & "'"
  DataGridView1.DataSource = dv
End Sub
```

Die gefilterte Ausgabe sehen Sie in Abbildung 9.21.

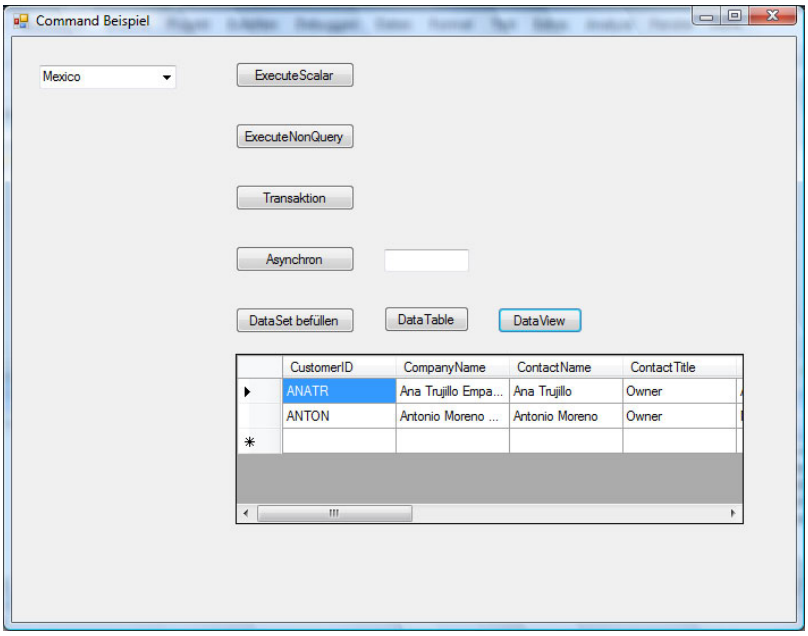


Abbildung 9.21 DataView mit gesetztem Filter

Relations

Mittels Relations können innerhalb eines `DataSet` Beziehungen zwischen Tabellen definiert werden. Da ein `DataSet` kein Wissen über die Herkunft der Daten hat, können auch keine Beziehungen zwischen den Tabellen auf der ursprünglichen Datenbank automatisch im `DataSet` übernommen werden. Mit `Relations` kann man Beziehungen, welche die Tabellen in der Datenbank haben, auch im `DataSet` nachbauen.

In unserem Beispiel will ich noch eine zweite `DataGridView` auf das Formular ziehen und darin Bestellungen der Kunden anzeigen. Durch eine definierte Relation synchronisiert sich die Anzeige beim Blättern in den Kunden automatisch.

Zu diesem Zweck habe ich den `Form_Load()`-Ereignishandler wie in Listing 9.12 umgeschrieben.

Listing 9.12 Relationen zwischen Tabellen

```
Private Sub frmRelation_Load(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load

  Dim da As New SqlDataAdapter("Select * from Customers", con)
  da.Fill(ds, "Customers")
  DataGridView1.DataSource = ds
  DataGridView1.DataMember = "Customers"

  Dim daOrder As New SqlDataAdapter( _
      "Select * from Orders", con)
  daOrder.Fill(ds, "Orders")
```

```
  ds.Relations.Add("RelCustomerOrders", _
  ds.Tables("Customers").Columns("CustomerId"), _
  ds.Tables("Orders").Columns("CustomerId"))

  DataGridView2.DataSource = ds
  DataGridView2.DataMember = "Customers.RelCustomerOrders"
End Sub
```

Listing 9.12 (Forts.) Relationen zwischen Tabellen

In diesem Beispiel befülle ich eine zweite Tabelle *Orders* in meinem `DataSet`.

Dann füge ich der `Relations`-Auflistung meines `DataSet` eine neue Relation hinzu. Dabei gebe ich den Relationsnamen an sowie die beiden Schlüsselspalten aus der Mastertabelle und der Beziehungstabelle. Optional steht noch ein vierter Parameter zur Verfügung, der überprüfen kann, ob eine referenzielle Integrität zwischen den beiden Tabellen bestehen soll, also ob es für jeden Eintrag in der *Orders*-Tabelle einen passenden Eintrag in der *Customers*-Tabelle geben muss.

In diesem Fall können wir uns ziemlich sicher sein, dass die Sache funktioniert, da wir eine Relation von der Datenbank abbilden. Da ein `DataSet` jedoch ein providerunabhängiges Objekt ist, könnte es durchaus auch möglich sein, dass die Daten aus unterschiedlichen Datenquellen kommen. Die Wahrscheinlichkeit, dass hier Sätze nicht zusammenpassen, ist relativ hoch, und dann können Sie mit dem Parameterwert `False` die referenzielle Integritätsüberprüfung für diese Beziehung abschalten (der Standardwert ist `True`).

Dann setze ich die `DataSource` der zweiten `DataGridView` auch wieder auf den gemeinsamen `DataSet`. Damit sich beide Grids synchronisieren, muss zwingend dieselbe `DataSource` angegeben werden. Als `DataMember` gebe ich die Relation an. Etwas gewöhnungsbedürftig ist wohl die Tatsache, dass sich die Relation von der Mastertabelle ableitet und somit in der Form `Mastertabelle.Relationsname ("Customers.RelCustomerOrders")` geschrieben werden muss.

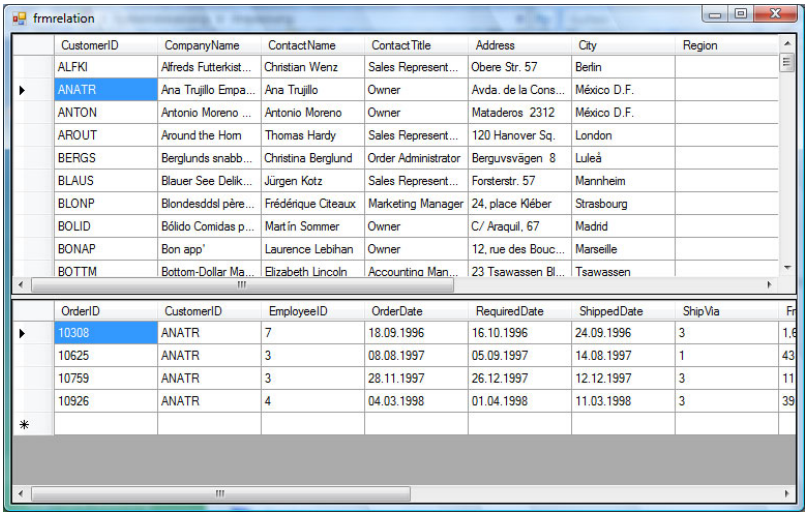


Abbildung 9.22 Relationsdarstellung mit zwei dataGridViews

Zum Abschluss dieses Abschnitts will ich Ihnen noch zeigen, wie Sie die Datensätze einer Mastertabelle inklusive der untergeordneten Datensätze durchlaufen können.

Dazu verwenden wir zwei ineinander geschachtelte `For Each`-Schleifen. Die äußere durchläuft sämtliche Datensätze der Kundentabelle und in der inneren Schleife werden alle abhängigen Zeilen zur aktuellen Zeile, die aus der angegebenen Beziehung stammen, ausgegeben. Den zugehörigen Programmcode sehen Sie in Listing 9.13.

Listing 9.13 Durchlaufen der Datensätze in einer Relation

```
Private Sub btnrelation_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnrelation.Click

  For Each dr As DataRow In ds.Tables("Customers").Rows

    Debug.WriteLine(dr("CustomerId").ToString)
    For Each dr2 As DataRow In _
            dr.GetChildRows("RelCustomerOrders")

      Debug.WriteLine("  " & dr2("OrderDate").ToString)
    Next
  Next
End Sub
```

Die Ausgabe des Programmcodes aus Listing 9.13 sehen Sie in Abbildung 9.23.

Abbildung 9.23 Ausgabe der Sätze mit abhängigen Zeilen, die durchlaufen werden

9.5 Datenbindung ohne Designer

Im letzten Abschnitt haben wir ja schon programmatische Datenbindung an das `DataGridView` durchgeführt. Hier will ich noch Datenbindung an weiteren Steuerelementen zeigen, nämlich an `TextBoxen` und an eine `ComboBox`.

Zu diesem Zweck lege ich ein neues Projekt vom Typ Windows Forms-Anwendung an. Ich verwende dabei dieselben Verbindungseinstellungen wie in den vorherigen Projekten.

Dabei will ich in einer `ComboBox` alle deutschen Kunden anzeigen.

Ich befülle also eine `DataTable` und binde die Tabelle über die Eigenschaft `DataSource` an die `ComboBox`. Danach setze ich über die Eigenschaft `DisplayMember` die Spalte, die in der Liste angezeigt werden soll. Über die Eigenschaft `ValueMember` könnte ich eine Schlüsselspalte an die Liste binden.

Der große Unterschied zu früheren Versionen der `ComboBox` besteht darin, dass die gesamte `DataRow` zur `ComboBox` hinzugefügt wird. Man kann also nicht nur auf die `DisplayMember`- und `ValueMember`-Eigenschaft zugreifen, sondern auf jede beliebige Spalte der Datenzeile.

Dies will ich ganz kurz demonstrieren, indem ich bei einer veränderten Auswahl in der Liste in einer `MessageBox` einen beliebigen Spaltenwert anzeige. Die Eigenschaft `SelectedItem` der `ComboBox` gibt mir ein Objekt zurück, das ich in eine `DataRowView` umwandle. Anschließend rufe ich den entsprechenden Spaltenwert ab.

Den Ereignishandler habe ich dabei nicht mit der Handles-Anweisung fest angegeben, sondern ich füge diesen dynamisch am Ende der `Form_Load()`-Methode hinzu. Somit ist gewährleistet, dass nicht schon beim Laden des Formulars die Meldung auf dem Bildschirm erscheint.

Den zugehörigen Programmcode sehen Sie in Listing 9.14.

Listing 9.14
Datenbindung an eine ComboBox

```
Imports System.Data.SqlClient
Imports System.Data.Configuration

Public Class Form1

  Private con As New SqlConnection _
    (ConfigurationManager.ConnectionStrings _
    ("NorthwindConnection").ConnectionString)

  Private dt As New DataTable

  Private Sub Form1_Load(ByVal sender As System.Object, _
     ByVal e As System.EventArgs) Handles MyBase.Load

    Dim da As New SqlDataAdapter("Select * from customers " & _
       "where Country = 'Germany'", con)
    da.Fill(dt)
    ComboBox1.DataSource = dt
    ComboBox1.DisplayMember = "Companyname"
    ComboBox1.ValueMember = "CustomerId"
    AddHandler ComboBox1.SelectedIndexChanged, _
        AddressOf ComboBox1_SelectedIndexChanged
  End Sub

  Private Sub ComboBox1_SelectedIndexChanged _
    (ByVal sender As System.Object, ByVal e As EventArgs)

    Dim dr As DataRowView = _
      CType(ComboBox1.SelectedItem, DataRowView)

    MessageBox.Show(dr("ContactName").ToString)
  End Sub
End Class
```

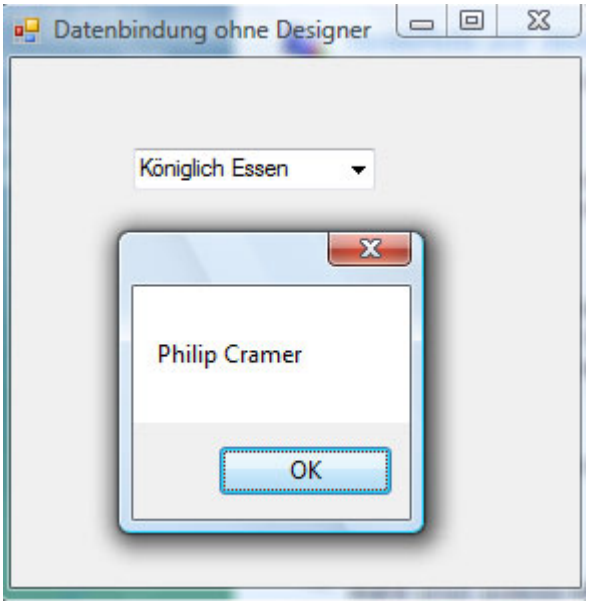


Abbildung 9.24
Datenbindung an eine ComboBox

Im nächsten Schritt möchte ich noch die Datenbindung an TextBoxen zeigen. Ich ziehe einfach exemplarisch drei TextBoxen auf das Formular.

Dazu füge ich die folgenden drei Zeilen in die `Form_Load()`-Methode ein.

```
tbCity.DataBindings.Add("Text", dt, "City")
tbCountry.DataBindings.Add("Text", dt, "Country")
tbPhone.DataBindings.Add("Text", dt, "Phone")
```

Dadurch werden die entsprechenden Spalten in den Textfeldern angezeigt. Ich füge mit diesen Anweisungen der `DataBindings`-Auflistung der jeweiligen Textfelder eine neue Datenbindung hinzu. Theoretisch kann ich jede Eigenschaft eines Steuerelements mit einer Datenbindung versehen, im obigen Beispiel wurde die Texteigenschaft gewählt, was wohl auch der häufigste Fall in der Praxis sein wird. Als zweiter Parameter wird die Datenquelle angegeben, in diesem Fall die `DataTable dt`. Dadurch, dass die `ComboBox` dieselbe `DataSource` besitzt, werden sich die Inhalte der `TextBox` bei einer anderen Auswahl der Liste automatisch anpassen. Als letzter Parameter wird schließlich noch die anzuzeigende Spalte als `DataMember` angegeben.

Wenn wir jetzt nicht die Liste hätten, könnten wir eigentlich nicht durch den Datenbestand blättern. Deswegen lege ich vier Buttons auf mein Formular, um durch den Datenbestand, wie in Abbildung 9.25 zu sehen, blättern zu können.

Für das Blättern verwende ich die `BindingContext`-Auflistung des Formulars. Für jede gebundene Datenquelle wird innerhalb des Formulars ein `BindingContext` zu dieser Auflistung hinzugefügt. Ich wähle also den `BindingContext` für die `DataTable dt` und passe die `Position`-Eigenschaft an. Sie können über diesen `BindingContext` auch neue Sätze anlegen oder löschen.

Listing 9.15 zeigt den entsprechenden Programmcode.

Listing 9.15
Verändern der Eigenschaft Position eines BindingContext

```
Private Sub btnFirst_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnFirst.Click
    Me.BindingContext(dt).Position = 0
End Sub

Private Sub btnPrevious_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnPrevious.Click
    Me.BindingContext(dt).Position -= 1
End Sub

Private Sub btnNext_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnNext.Click
    Me.BindingContext(dt).Position += 1
End Sub

Private Sub btnLast_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnLast.Click
    Me.BindingContext(dt).Position = dt.Rows.Count - 1
End Sub
```

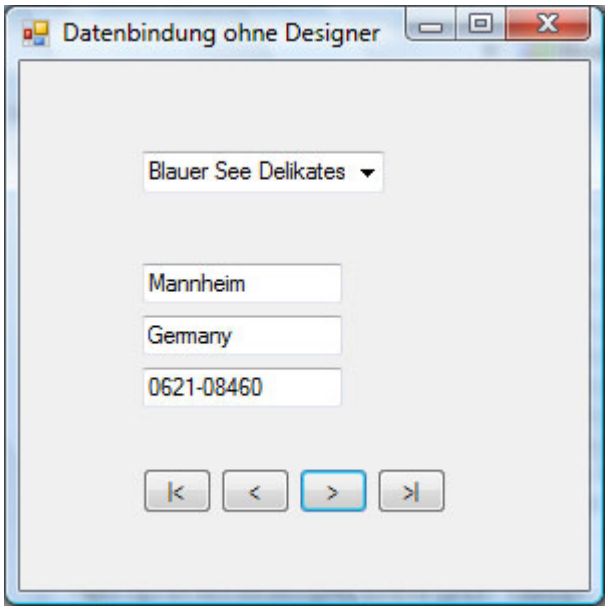


Abbildung 9.25
Blätterfunktion durch den Datenbestand

9.6 Datenbindung mit ObjectSources

Eine alternative Möglichkeit der Datenbindung bietet die Bindung an Objektklassen, zum Beispiel aus Ihrer Mittelschicht in mehrschichtigen Applikationen. Auch wenn das im eigentlichen Sinne nichts mit ADO.NET zu tun hat, will ich Ihnen zumindest ein kleines Beispiel präsentieren.

Dafür habe ich eine Klasse `Player` geschrieben, die Sie in Listing 9.16 sehen.

Listing 9.16 Listing der Klasse Player

```
Public Class Player
    Private mFirstName As String
    Private mLastName As String
    Private mPosition As String

    Public Property FirstName() As String
        Get
            Return mFirstName
        End Get
        Set(ByVal value As String)
            mFirstName = value
        End Set
    End Property

    Public Property LastName() As String
        Get
            Return mLastName
        End Get
        Set(ByVal value As String)
            mLastName = value
        End Set
    End Property

    Public Property Position() As String
        Get
            Return mPosition
        End Get
        Set(ByVal value As String)
            mPosition = value
        End Set
    End Property

End Class
```

Die Klasse besteht lediglich aus drei Eigenschaften *FirstName*, *LastName* und *Position*.

In einer zweiten Klasse `DataAccess` biete ich die Schnittstelle zur Objektbindung. Dabei gebe ich eine generische Auflistung von `Player`-Objekten in einer Methode `SelectPlayers()` zurück. Die `Player`-Auflistung `Starters` in `SelectPlayers()` erzeuge ich für das Beispiel rein durch Code. Für das Zurückschreiben der Daten habe ich drei weitere Methoden `Update()`, `Delete()` und `Insert()` hinzugefügt, die ich jedoch nicht ausprogrammiert habe. Sie sehen dies in Listing 9.17.

Listing 9.17 Die Klasse DataAccess für ObjectBinding

```
Imports System.Collections.Generic
Public Class DataAccess
  Private Starters As List(Of Player)

  Public Function SelectPlayers() As List(Of Player)

    Starters = New List(Of Player)
```

Listing 9.17 (Forts.)
Die Klasse DataAccess für ObjectBinding

```
    Dim p1 As New Player() With _
       {.FirstName = "Jimmy", .LastName = "Robertson", _
        .Position = "Quarterback"}
    Starters.Add(p1)

    Dim p2 As New Player() With _
       {.FirstName = "Andrew", .LastName = "Blakley", _
        .Position = "Wide Receiver"}
    Starters.Add(p2)
    …
    Dim p7 As New Player() With _
       {.FirstName = "Christian", .LastName = "Früchtl", _
        .Position = "Right Guard"}
        …
    Starters.Add(p11)

    Return Starters
  End Function

  Public Sub Delete(ByVal LastName As String)
    'Funktionalität für das Löschen
  End Sub

  Public Sub Update(ByVal c As Player)
     'Funktionalität für Update
  End Sub

  Public Sub Insert(ByVal c As Player)
     'Funktionalität für Insert
  End Sub
End Class
```

Nachdem ich die beiden Klassen implementiert habe, ziehe ich auf das Formular eine `DataGridView` und füge den Programmcode aus Listing 9.18 hinzu.

Genauso gut hätte ich beide Klassen auch als Klassenbibliothek kompilieren und in der Windows-Anwendung einen Verweis auf die generierte Klassenbibliothek setzen können.

Listing 9.18
Objektdatenbindung

```
Public Class Form1

  Private Sub Form1_Load(ByVal sender As System.Object, _
     ByVal e As System.EventArgs) Handles MyBase.Load

    Dim myObjectBinder As New DataAccess

      DataGridView1.DataSource = myObjectBinder.SelectPlayers
    End Sub
End Class
```

In Listing 9.18 erstelle ich erst einmal eine Instanz meiner Middleware-Klasse `DataAccess` und setze dann als `DataSource` des `DataGridView` den Rückgabewert der Methode `SelectPlayers()` der Klasse `DataAccess`. Die Bildschirmausgabe sehen Sie in Abbildung 9.26.

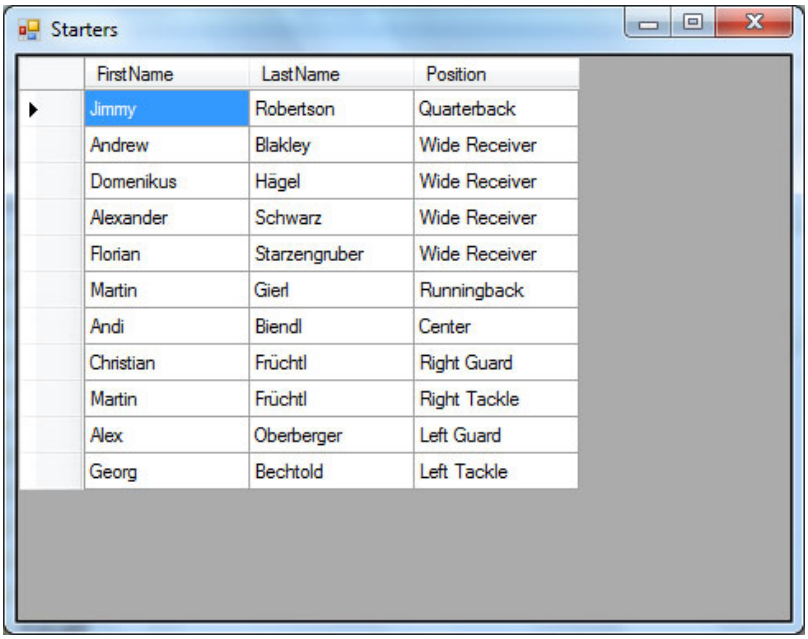


Abbildung 9.26 Ausgabe des ObjektBinding

9.7 Weitere Features in ADO.NET

In diesem Abschnitt will ich Ihnen noch weitere wichtige Möglichkeiten von ADO.NET, die bislang in diesem Kapitel noch nicht angesprochen wurden, vorstellen. Der Großteil dieser Features ist jedoch nur für den Microsoft SQL Server verfügbar.

Fangen wir jedoch mit den für alle Provider verfügbaren Neuerungen an.

9.7.1 Providerunabhängiges Programmieren

Für Software-Entwickler, die sich nicht auf ein bestimmtes Datenbanksystem festlegen können, was bei Standardsoftware sehr oft der Fall sein kann, war es bislang sehr aufwändig, providerunabhängige Datenbankzugriffe zu implementieren. Im Grunde konnten alle Definitionen nur auf der Basis von Interfaces durchgeführt werden und an vielen Stellen im Programmcode müssen immer die entsprechenden Fallunterscheidungen durchgeführt werden.

In ADO.NET wurde eine `DbProviderFactory` eingeführt, die genau dieses Problem lösen soll. Über diese Factory werden die gewünschten `SqlConnection`-, `OracleConnection`- oder `OledbConnection`-Objekte erzeugt, wobei der gewünschte Provider über eine Konfigurationsdatei gesetzt werden kann.

Um dies zu demonstrieren, habe ich ein neues Projekt vom Typ Windows Forms-Anwendung angelegt und in der Konfigurationsdatei den bislang verwendeten `ConnectionString` für die Northwind-Datenbank hinterlegt, den ich hier noch mal kurz darstellen möchte:

```
<connectionStrings>
  <add
    name="NorthwindConnection"
      connectionString="Data Source=(local)\SqlExpress;
      Initial Catalog=Northwind;
      Integrated Security=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

Wie Sie sehen, gibt es für diesen `ConnectionString` auch ein eigenes Element `providerName`, das wir uns gleich zunutze machen werden. Denn darüber können wir steuern, aus welchem Namespace wir unsere providerabhängigen Objekte erstellen wollen.

Woher weiß aber jetzt die .NET-Laufzeitumgebung, für welchen String sie welche Klassen laden soll? Die Lösung ist, wie so oft, in der *machine.config* hinterlegt. In der *machine.config* gibt es eine eigene Sektion `DbProviderFactories`, in der alle installierten .NET DataProvider enthalten sind, wie Sie in Listing 9.19 sehen.

Listing 9.19
.NET DataProvider in der machine.config

```
<system.data>
  <DbProviderFactories>
    <add
      name="Odbc Data Provider"
      invariant="System.Data.Odbc"
      description=".Net Framework Data Provider for Odbc"
      type="System.Data.Odbc.OdbcFactory,
        System.Data, Version=4.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089"
    />
    <add
      name="OleDb Data Provider"
      invariant="System.Data.OleDb"
      description=".Net Framework Data Provider for OleDb"
      type="System.Data.OleDb.OleDbFactory,
        System.Data, Version=4.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089"
    />
    <add
      name="OracleClient Data Provider"
      invariant="System.Data.OracleClient"
      description=".Net Framework Data Provider for Oracle"
      type="System.Data.OracleClient.OracleClientFactory,
        System.Data.OracleClient,
        Version=4.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089"
    />
    <add
      name="SqlClient Data Provider"
      invariant="System.Data.SqlClient"
```

Listing 9.19 (Forts.) .NET DataProvider in der machine.config

```
      description=".Net Framework Data Provider for SqlServer"
      type="System.Data.SqlClient.SqlClientFactory,
            System.Data, Version=4.0.0.0, Culture=neutral,
        PublicKeyToken=b77a5c561934e089"
    />
    <add
      name="SQL Server CE Data Provider"
      invariant="Microsoft.SqlServerCe.Client"
      description=".NET Framework Data Provider for
                   Microsoft SQL Server 2005 Mobile Edition"
      type="Microsoft.SqlServerCe.Client.SqlCeClientFactory,
        Microsoft.SqlServerCe.Client,
        Version=9.0.242.0, Culture=neutral,
        PublicKeyToken=89845dcd8080cc91"
    />
    <add
      name="Microsoft SQL Server Compact Data Provider"
      invariant="System.Data.SqlServerCe.3.5"
      description=".NET Framework Data Provider
               for Microsoft SQL Server Compact"
      type="System.Data.SqlServerCe.SqlCeProviderFactory,
        System.Data.SqlServerCe, Version=3.5.0.0,
        Culture=neutral, PublicKeyToken=89845dcd8080cc91"
    />
  </DbProviderFactories>
</system.data>
```

Hier sehen Sie eine Auflistung aller standardmäßig installierten Provider. Das Mapping zum passenden Provider erfolgt letztendlich über das Element `invariant` und es wird der unter `type` angegebene Typ geladen.

Als Erstes möchte ich gerne alle in die *machine.config* eingetragenen DataProvider ausgeben.

Dazu füge ich eine mehrzeilige `TextBox` auf mein Formular und schreibe in der `Form_Load()`-Methode die entsprechenden Einträge in die `TextBox`.

Meinem Formular füge ich außerdem eine Imports-Anweisung auf den Namespace `System.Data.Common` hinzu. Innerhalb dieses Namespace befinden sich alle benötigten Klassen für providerunabhängiges Programmieren.

Die statische Methode `GetFactoryClasses()` der Klasse `DbProviderFactories` befüllt mir eine `DataTable`, über die ich alle Einträge in die `TextBox` einfügen kann.

Listing 9.20 Auflistung aller installierten DbProviderfactories

```
Imports System.Data.Common
Imports System.Configuration
Private Sub Form1_Load(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load

  Dim dt As DataTable = DbProviderFactories.GetFactoryClasses

  For Each dr As DataRow In dt.Rows
    tbProvider.Text &= dr("Name").ToString & vbCrLf & vbTab _
      dr("InvariantName").ToString & vbCrLf
  Next
End Sub
```

So, nun will ich im Formular noch einen DataAdapter definieren, der eine Tabelle an eine ComboBox bindet. Diese Aufgabe soll dabei ohne Angabe eines spezifischen Connection- oder DataAdapter-Objekts gelöst werden.

Ich füge dem Formular noch eine Schaltfläche zum Erzeugen der Objekte und zum Laden der Daten sowie eine ComboBox zum Darstellen der Daten hinzu.

Den Programmcode unter der Laden-Schaltfläche sehen Sie in Listing 9.21.

Listing 9.21
Datenzugriff mit providerunabhängigem Code

```
Private Sub btnLaden_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnLaden.Click

  Dim factory As DbProviderFactory = _
    DbProviderFactories.GetFactory _
    (ConfigurationManager.ConnectionStrings _
    ("NorthwindConnection").ProviderName)

  Dim con As DbConnection =  factory.CreateConnection

  con.ConnectionString = ConfigurationManager. _
    ConnectionStrings("NorthwindConnection").ConnectionString

  Dim da As DbDataAdapter = factory.CreateDataAdapter
  da.SelectCommand = factory.CreateCommand
  da.SelectCommand.Connection = con
  da.SelectCommand.CommandText = "Select * from Customers"
  Dim dt As New DataTable
  da.Fill(dt)
  cboDaten.DataSource = dt
  cboDaten.DisplayMember = "Companyname"
End Sub
```

Zuerst generiere ich eine passende DbProviderFactory aufgrund des Eintrags providerName im ConnectionString innerhalb der Konfigurationsdatei.

Diese Factory erstellt mir dann die passende Connection, deren ConnectionString wiederum aus der Konfigurationsdatei gelesen wird.

Die Factory ist dann auch noch zur Erstellung des DataAdapter und der benötigten Command-Objekte zuständig. Der erzeugte DataAdapter befüllt eine DataTable, die an die ComboBox gebunden wird, und schon ist die providerunabhängige Anwendung fertig.

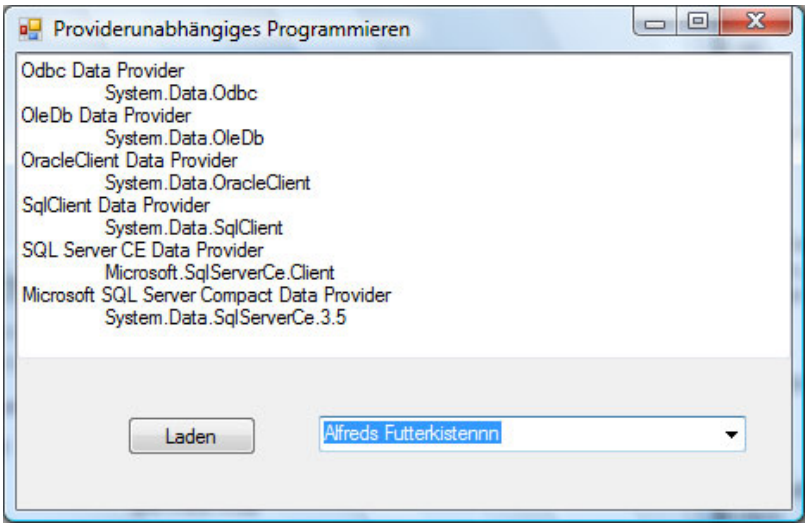


Abbildung 9.27 Ausgabe der Anwendung »Providerunabhängiges Programmieren«

9.7.2 Nullable Types

Wertetypen, wie zum Beispiel `Integer`, `DateTime` und `Double`, werden mit Standardwerten vorinitialisiert. Das bedeutet, sie besitzen immer einen Wert, und es kann ihnen kein NULL-Wert zugewiesen werden. In Datenbanken sind aber sehr oft für Spalten von diesem Datentyp NULL-Werte nötig. Wenn Sie zum Beispiel ein Geburtsdatum einer Person nicht wissen, wollen Sie wohl an dieser Stelle lieber einen NULL-Wert in die Datenbank eintragen als den Standardwert 01.01.0001.

Auch bei der Zuweisung von Datenbankinhalten an Wertetypen musste man immer überprüfen, ob der Datenbankwert nicht NULL ist, weil ansonsten ein Laufzeitfehler bei der Zuweisung aufgetreten ist.

Aus diesem Grunde wurden bereits in .NET 2.0 sogenannte Nullable Types eingeführt, diese Nullable Types stellen dabei generische Typen dar.

Mit diesen Typen können auch NULL-Werte an Wertetypen zugewiesen werden, oder ganz konkret gesagt, diesen Typen kann `Nothing` zugewiesen werden.

Die Definition eines Nullable Integers würde so aussehen:

```
Dim iWert as Integer?
```

oder etwas ausführlicher auch so:

```
Dim iWert as Nullable(Of Integer)
```

Dieser Variablen kann tatsächlich problemlos auch ein NULL-Wert aus der Datenbank zugewiesen werden oder sie kann direkt mit folgender Anweisung

```
iWert = Nothing
```

auf `Nothing` gesetzt werden.

9.7.3 SQLBulkCopy

Stellen Sie sich vor, Sie wollen große Datenmengen zu einem anderen Server kopieren. Als Erstes würde mir hier eine *Select*-Anweisung mit einer anschließenden *Insert Into*-Anweisung zum neuen Datenbankserver einfallen. Das würde aber sehr lange dauern und eine fehlerlose Ausführung ist auch nicht garantiert.

Mit der Klasse `SqlBulkCopy` hat Microsoft in ADO.NET jedoch eine Möglichkeit geschaffen, auch große Datenmengen ohne größere Umstände zu kopieren.

Das folgende Beispiel zeigt, wie zunächst eine Verbindung zur vorhandenen Datenbank aufgebaut wird. Danach werden alle Daten der vorhandenen Datenbank mit dem `SqlCommand` in einen `SqlDataReader` eingelesen und ebenfalls eine Verbindung zur neuen Datenbank aufgebaut. Die Klasse `SqlBulkCopy` definiert die Verbindung zum Ziel. Mit der Methode `WriteToServer()` wird anschließend der Transfer zum Server gestartet.

Einen Beispielprogrammcode zum Massenkopieren sehen Sie in Listing 9.22.

Listing 9.22 Massenkopieren mittels SqlBulkCopy

```
Public Sub bulkcopy()
  Dim sqlConnectionQuelle As New SqlConnection("…")
  Dim sqlConnectionZiel As New SqlConnection("…")

  Dim sqlcommandBackup As New SqlCommand _
    ("select * from Customers", sqlConnectionQuelle)
  'Verbindungen öffnen
  sqlConnectionQuelle.Open()
  sqlConnectionZiel.Open()

  Dim sqlReader As SqlDataReader = _
    sqlcommandBackup.ExecuteReader

  'Kopieren
  Dim sqlbulkcopy As New SqlBulkCopy(sqlConnectionZiel)

  sqlbulkcopy.DestinationTableName = "Customers"
  sqlbulkcopy.WriteToServer(sqlReader)

  sqlConnectionQuelle.Close()
  sqlConnectionZiel.Close()
End Sub
```

Achtung

SqlBulkCopy ist nur mit dem Microsoft SQL Server möglich.

9.7.4 MARS – MultipleActiveResultsets

Früher war es nicht möglich, mehrere DataReader gleichzeitig zu öffnen. Jeder DataReader belegte eine Verbindung exklusiv, das bedeutete, solange der Reader geöffnet war, konnte keine andere Anweisung über diese Verbindung ausgeführt werden. ADO.NET ab der Version 2.0 hat zumindest für den Microsoft SQL Server ab der Version 2005 die Möglichkeit geschaffen, diese Restriktion zu überwinden. Das Zauberwort dazu heißt MARS – MultipleActiveResultsets.

Damit Sie diese Funktionalität nutzen können, muss der `ConnectionString` zum SQL Server einen zusätzlichen Eintrag `MultipleActiveResultSets=True` enthalten.

Nun können problemlos zwei und mehr Reader gleichzeitig geöffnet werden. Die Abarbeitung erfolgt dabei synchron.

9.8 XML

Seit 1998 ist XML (eXtensible Markup Language) ein standardisiertes Datenaustauschformat des W3C. Nachdem sich XML in den letzten Jahren wie ein Lauffeuer verbreitet hat, kann man es heute bereits als wichtige Brücke zwischen dem Austausch von relationalen Daten und Dokumenten sehen. Mit XML ist es möglich, auch Daten, die auf verschiedenen Datenbanksystemen basieren, miteinander auszutauschen. Microsoft hat den Lauf der Zeit erkannt und den Namespace `System.Xml` ebenso erweitert.

Als Entwickler sollten Sie sich um das Thema XML kümmern, da gerade mit XML Probleme, die beim Lösen von Daten- und Plattformunabhängigkeit auftreten, behoben werden können.

Achtung

Das XML-Format, das in .NET verwendet wird, ist das standardisierte XML 1.0. Auf eine weitere Verwendung und Weiterentwicklung von MSXML hat Microsoft verzichtet.

9.8.1 System.Xml

Der Namespace `System.XML` beinhaltet alle Klassen, die für den Umgang mit XML-Dateien benötigt werden.

Prinzipiell gibt es im .NET Framework zwei Möglichkeiten, auf XML-Daten zuzugreifen.

Zum einen können Sie die gesamte XML-Datei mit dem `XmlDocument`-Objekt komplett in den Speicher laden und den Dokumentenbaum aus Objekten wie `XmlElement` oder `XmlAttribute` aufbauen.

Die zweite Möglichkeit besteht darin, die Daten zeilenweise zu lesen, zu schreiben oder zu validieren. Dabei stehen Ihnen die Objekte `XmlReader` und `XmlWriter` zur Verfügung. Diese Methode ist zwar nicht so komfortabel wie das gesamte Laden des Dokuments, jedoch bei weitem speicherschonender als das Arbeiten mit `XmlDocument`.

In diesem Abschnitt lernen Sie die grundlegenden Merkmale und Klassen des `System.Xml`-Namespace in Bezug auf die beiden genannten Methoden kennen.

9.8.2 Lesen und Navigieren mit XmlDocument

Das Lesen und Ausgeben eines XML-Dokuments kann relativ simpel erfolgen. So liest bereits folgender Code, wenn auch ungeordnet, den gesamten Inhalt einer XML-Datei ein und gibt diesen in einer Konsole aus.

Listing 9.23
Einfaches Lesen eines XML-Dokuments

```
Dim doc As XmlDocument = New XmlDocument()
doc.Load("Customers.xml")
doc.Save(Console.Out)
```

Mit `System.Xml.XPath.XPathDocument` wurde eine Klasse geschaffen, die sowohl Daten einliest als auch durch XML-Dokumente navigieren kann. Die Navigation durch ein XML-Dokument wird mit `XPathDocument` zum Kinderspiel. So sind die Funktionen, welche die Navigation durch die Klasse ermöglichen, fast selbst erklärend. Hier findet immer das Präfix `Move` Verwendung. So können Sie mit `MoveToRoot()` zum Anfang des Dokuments springen, mit `MoveToNext()` zum nächsten Element und mit `MoveToID()` zu einem bestimmten Element, das mit einer bestimmten ID gekennzeichnet ist.

Als Beispiel wurde die Tabelle *Customers* aus der Northwind-Datenbank in das XML-Format exportiert. Sie liegt nun mit folgendem Inhalt vor:

Listing 9.24
Die Tabelle Customers im XML-Format

```
<?xml version="1.0" standalone="yes"?>
<CustomerData>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <CompanyName>Alfreds Futterkisten</CompanyName>
    <ContactName>Maria Anders</ContactName>
    <ContactTitle>Sales Representative</ContactTitle>
    <Address>Obere Str. 57</Address>
    <City>Berlin</City>
    <Region />
    <PostalCode>12209</PostalCode>
    <Country>Germany</Country>
    <Phone>030-0074321</Phone>
    <Fax>030-0076545</Fax>
  </Customers>
  <Customers>
    <CustomerID>ANATR</CustomerID>
    <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
    <ContactName>Ana Trujillo</ContactName>
    <ContactTitle>Owner</ContactTitle>
    <Address>Avda. de la Constitución 2222</Address>
    <City>México D.F.</City>
    <PostalCode>05021</PostalCode>
    <Country>Mexico</Country>
    <Phone>(5) 555-4729</Phone>
    <Fax>(5) 555-3745</Fax>
  </Customers>
...
```

Listing 9.24 (Forts.) Die Tabelle Customers im XML-Format

```
  <Customers>
    <CustomerID>WOLZA</CustomerID>
    <CompanyName>Wolski  Zajazd</CompanyName>
    <ContactName>Zbyszek Piestrzeniewicz</ContactName>
    <ContactTitle>Owner</ContactTitle>
    <Address>ul. Filtrowa 68</Address>
    <City>Warszawa</City>
    <PostalCode>01-012</PostalCode>
    <Country>Poland</Country>
    <Phone>(26) 642-7012</Phone>
    <Fax>(26) 642-7012</Fax>
  </Customers>
</CustomerData>
```

Im folgenden Beispiel werden alle Knoten und deren Inhalte durchgegangen und in der Konsole ausgegeben. Hier findet der `XPathNavigator` Verwendung, der es ermöglicht, mit `MoveTo()` alle Elemente zu überprüfen. Sobald ein neuer Knoten gefunden wurde, werden sowohl der Name als auch der Wert ausgegeben.

Listing 9.25 Alle Knoten und Inhalte der XML-Datei

```
Public Sub lesexml()
  'XML-Datei laden
   Dim xmldoc As New XPath.XPathDocument("Customers.xml")
   Dim i As Integer = 1

  'Navigator initialisieren, um zu navigieren
  Dim nav As System.Xml.XPath.XPathNavigator = _
    xmldoc.CreateNavigator

  'Durch Knoten navigieren
  nav.MoveToFirstChild()
  nav.MoveToFirstChild()

  'Alle Kontakte ausgeben
  Do
    Console.WriteLine(i.ToString() & ". Knoten:")
    nav.MoveToFirstChild()

    'Die Daten der Kunden ausgeben
     Do
       Console.WriteLine(nav.Name & ": " & nav.Value)
     Loop While nav.MoveToNext
     Console.WriteLine()
     i += 1
     nav.MoveToParent()
  Loop While nav.MoveToNext
End Sub
```

Einen Teil der Ausgabe, nachdem die Methode `lesexml()` einmal aufgerufen wurde, sehen Sie in Abbildung 9.28:

In diesem Beispiel kann man bereits die Struktur eines XML-Dokuments erkennen.

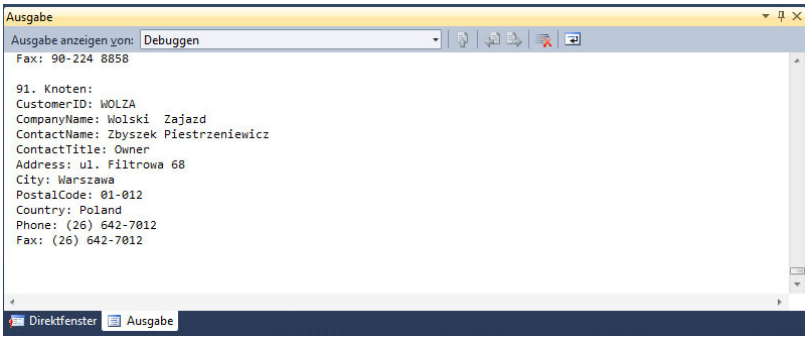


Abbildung 9.28
Ausgabe der XML-Navigation

9.8.3 XML erzeugen

Im nächsten Beispiel will ich eine komplett neue XML-Datei erzeugen. Dabei demonstriere ich einmal den `XmlWriter` und zum zweiten das `XmlDocument`.

XmlWriter

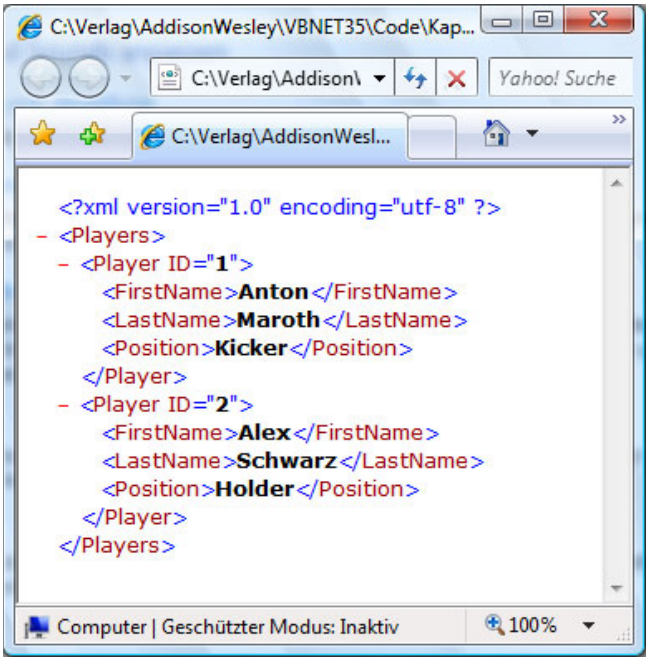
Mit dem `XmlWriter` können Sie Zeile für Zeile in eine XML-Datei schreiben. Eine Navigation während des Schreibens ist nicht möglich.

Folgendes Beispiel erzeugt ein komplett neues XML-Dokument.

Listing 9.26
XmlWriter im Einsatz

```
Public Sub xmlSchreibenMitXmlWriter()
  Dim writer As XmlWriter
  'Datei erzeugen
  writer = XmlWriter.Create("Players.xml")
  writer.WriteStartDocument()
  'Start-Element schreiben
  writer.WriteStartElement("Players")

  'Neuen Spieler anlegen
  writer.WriteStartElement("Player")
  writer.WriteAttributeString("ID", "1")
  writer.WriteElementString("FirstName", "Anton")
  writer.WriteElementString("LastName", "Maroth")
  writer.WriteElementString("Position", "Kicker")
  writer.WriteEndElement()

  writer.WriteStartElement("Player")
  writer.WriteAttributeString("ID", "2")
  writer.WriteElementString("FirstName", "Alex")
  writer.WriteElementString("LastName", "Schwarz")
  writer.WriteElementString("Position", "Holder")
  writer.WriteEndElement()

  'Datei schließen
  writer.WriteEndDocument()
  writer.Close()
End Sub
```

Abbildung 9.29
Ausgabe der erstellten XML-Datei im Internet Explorer



XmlDocument

Im zweiten Schritt erzeugen wir mittels geeigneter Objekte den kompletten Objektbaum im Speicher und schreiben am Ende die Daten in eine Datei. Im Gegensatz zum `XmlWriter` sehen Sie, dass Sie Knoten immer an andere Knoten anhängen und sich nicht um das korrekte Schreiben der zugehörigen Endelemente kümmern müssen.

Listing 9.27
Erzeugen einer Datei mittels XDocument

```
Private Sub xmlSchreibenMitXmlDocument()
  Dim doc As New XmlDocument()
  Dim elRoot As XmlElement = doc.CreateElement("Players")
  doc.AppendChild(elRoot)

  Dim elPlayer As XmlElement = doc.CreateElement("Player")
  Dim att1 As XmlAttribute = doc.CreateAttribute("ID")
  att1.Value = "1"
  elPlayer.Attributes.Append(att1)

  Dim elFirstName As XmlElement = _
    doc.CreateElement("FirstName")
  elFirstName.InnerText = "Anton"
  Dim elLastName As XmlElement = doc.CreateElement("LastName")
  elLastName.InnerText = "Maroth"
  Dim elPosition As XmlElement = doc.CreateElement("Position")
  elPosition.InnerText = "Kicker"

  elPlayer.AppendChild(elFirstName)
  elPlayer.AppendChild(elLastName)
  elPlayer.AppendChild(elPosition)
```

```
  Dim elPlayer2 As XmlElement = doc.CreateElement("Player")
  Dim att2 As XmlAttribute = doc.CreateAttribute("ID")
  att2.Value = "2"
  elPlayer2.Attributes.Append(att2)

  Dim elFirstName2 As XmlElement = _
    doc.CreateElement("FirstName")
  elFirstName.InnerText = "Alex"
  Dim elLastName2 As XmlElement = doc.CreateElement("LastName")
  elLastName.InnerText = "Schwarz"
  Dim elPosition2 As XmlElement = doc.CreateElement("Position")
  elPosition.InnerText = "Holder"

  elPlayer2.AppendChild(elFirstName2)
  elPlayer2.AppendChild(elLastName2)
  elPlayer2.AppendChild(elPosition2)

  elRoot.AppendChild(elPlayer)
  elRoot.AppendChild(elPlayer2)

  doc.Save("players2.xml")
End Sub
```

Listing 9.27 (Forts.) Erzeugen einer Datei mittels XDocument

Diese beiden Beispiele haben Ihnen gezeigt, wie Sie eine komplett neue XML-Datei erstellen. Was ist aber, wenn Sie in eine bereits bestehende XML-Datei nur neue Werte einfügen möchten? Hier kommt wieder der `XPathNavigator` ins Spiel. Sie können mit der Funktion `MoveTo()` zu einem bestimmten Punkt im Dokument springen und schließlich mit `AppendChild()` neue Elemente in die XML-Datei einfügen.

9.8.4 Vorhandene Daten bearbeiten

Um bereits vorhandene Daten in einem XML-Dokument zu speichern bzw. Daten zu ändern, können wir wieder auf die Klasse `XmlDocument` zurückgreifen. Mit dieser kann man ein XML-Dokument in den Hauptspeicher laden und gezielt, beispielsweise mit `SelectNodes()` oder ganz speziell mit `SelectSingleNode()` auf Knoten zugreifen und diese verändern.

So wird zum Beispiel eine bereits vorhandene XML-Datei in den Hauptspeicher geladen. Der zu ändernde Knoten wird dann mit der Methode `SelectSingleNode()` der Klasse `XmlElement` ausgewählt und geändert. Anschließend wird das Dokument mit `Save()` gespeichert.

Das folgende Beispiel verdeutlicht, wie die Klasse `XmlDocument` arbeitet. Zunächst wird mit der Methode `Load()` die Datei geladen. Danach wird mit `SelectSingleNode()` das gewünschte Element referenziert. Die Abfrage erfolgt dabei mittels eines XPath-Ausdrucks.

Tipp

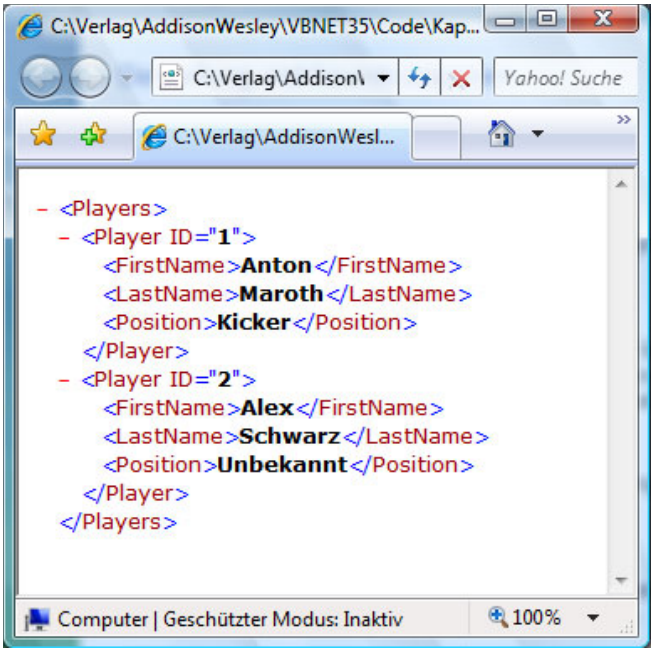
XPath ist eine Abfragesprache für XML. Im Internet finden Sie sehr umfangreiche XPath-Dokumentationen.

In diesem Beispiel wird die Position des zweiten Spielers auf `Unbekannt` geändert, bevor das Dokument schließlich mit `Save()` gespeichert und geschlossen wird.

Listing 9.28
XML-Elemente ändern

```
Public Sub edit()
  Dim doc As New XmlDocument()
  Dim el As XmlElement

  'Dokument laden
  doc.Load("players2.xml")

  'Knoten anhand der ID auswählen
  'Position-Knoten wählen
  el = CType(doc.SelectSingleNode _
      ("*//Player[@ID= 2]/Position"), XmlElement)
  'Wert ändern
  el.FirstChild.Value = "Unbekannt"
  'Wert speichern
  doc.Save("Players3.xml")
End Sub
```

Abbildung 9.30
Datei Players3.xml mit geändertem Inhalt



9.8.5 XML lesen mit XmlReader

Im nächsten Beispiel will ich Ihnen den `XmlReader` zeigen. Mit diesem `XmlReader` lese ich die gerade erzeugte *players.xml*-Datei zeilenweise aus. Listing 9.29 zeigt, wie die Datei geöffnet wird und die Daten knotenweise durchlaufen werden. In Abhängigkeit vom Knotentyp wird dann eine Meldung ausgegeben.

Listing 9.29
XmlReader im Einsatz

```
Public Sub LesenMitXmlReader()
  Dim elementName As String = ""
  Dim reader As XmlReader = XmlReader.Create("Players.xml")
  Do While reader.Read
    Select Case reader.NodeType
      Case XmlNodeType.Element
        elementName = reader.Name & ": "
      Case XmlNodeType.Text
      MessageBox.Show(elementName + reader.Value)
    End Select
  Loop
End Sub
```

9.8.6 XML validieren

Vielleicht haben Sie bereits mehr Erfahrungen im Umgang mit XML-Dateien sammeln können und Ihnen ist aufgefallen, dass das .NET Framework in Bezug auf XML-Dateien sehr genau ist. Sobald ein Tag fehlt oder nicht richtig geschlossen ist, verweigert es die gesamte Arbeit und beendet die Anwendung mit einer Fehlermeldung. Diese einheitliche Struktur wurde abgesegnet, um einen wirklich gültigen Standard zu schaffen. Um diesen Standard zu gewähren, hat jede XML-Datei ein Schema, früher kam DTD (Document Type Definition) zum Einsatz, dieses wurde aber durch die XML Schema Definition abgelöst. Beide werden jedoch noch vollständig und aus Kompatibilitätsgründen vom .NET Framework unterstützt. Ein Schema legt Richtlinien für ein XML-Dokument fest, die befolgt werden müssen, um das XML-Dokument zu verarbeiten. Um Fehler bei der Verarbeitung zu vermeiden, sollten Sie immer das zu bearbeitende XML-Dokument gegen ein Schema prüfen.

Das passende Schema für das bereits besprochene *Customers.xml*-Dokument würde wie folgt lauten:

Listing 9.30
Gültiges xsd-Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CustomerData">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded"
                    name="Customer">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CustomerID"
                          type="xs:string" />
              <xs:element name="CompanyName"
                          type="xs:string" />
              <xs:element name="ContactName"
                          type="xs:string" />
              <xs:element name="ContactTitle"
                          type="xs:string" />
              <xs:element name="Address"
                          type="xs:string" />
              <xs:element name="City"
```

Listing 9.30 (Forts.)
Gültiges xsd-Schema

```
                        type="xs:string" />
                <xs:element minOccurs="0"
                        name="Region"
                        type="xs:string" />
                <xs:element name="PostalCode"
                        type="xs:string" />
                <xs:element name="Country"
                        type="xs:string" />
                <xs:element name="Phone"
                        type="xs:string" />
                <xs:element name="Fax"
                        type="xs:string" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

Info

Sie können übrigens in Visual Studio 2010 direkt ein XML-Dokument validieren. Hierzu öffnen Sie das XML-Dokument. Das Visual Studio-Menü wird nun um den Eintrag XML erweitert und Sie können mit der Funktion Schema erstellen das Schema direkt erstellen.

Im .NET Framework geht die Erstellung eines Schemas auch direkt per Code. Listing 9.31 zeigt die Erstellung eines Schemas für unsere *players.xml*-Datei.

Listing 9.31
Schema aus einer XML-Datei per Code erstellen

```
Public Sub schemaerstellen()
  ' Schema auslesen
 Dim Inference As New Schema.XmlSchemaInference()
 Dim Schemaset As New Schema.XmlSchemaset()
 Schemaset = Inference.InferSchema( _
   XmlReader.Create("Players.xml"))

 'Schema erstellen
 Dim wr As XmlWriter = XmlWriter.Create _
   (New IO.StreamWriter("Players.xsd"))
 Dim schema As Schema.XmlSchema
 For Each schema  In Schemaset.Schemas()
   schema.Write(wr)
 Next schema
End Sub
```

Das erzeugte Schema sehen Sie in Abbildung 9.31.

Um die Datei nun gegen das Schema zu validieren – Sie können später gerne zum Testen mal kleine Fehler in die Datei einbauen – gehen Sie bitte wie folgt vor:

Als Erstes definieren Sie ein Objekt vom Typ `XmlReaderSettings`. Mit diesem Typ können Sie den Validierungstyp – wir wollen gegen ein Schema validieren – angeben und einen Ereignishandler definieren, der alle Fehler und Warnungen beim Validieren abfängt. Beim Ereignishandler `Fehlerbehandlung()` wird dann der Fehler in einem Meldungsfenster ausgegeben.

Danach wird das zugehörige Schema dem XML-Dokument hinzugefügt. Schließlich wird begonnen, das XML-Dokument zu lesen und somit gegen das Schema zu validieren.

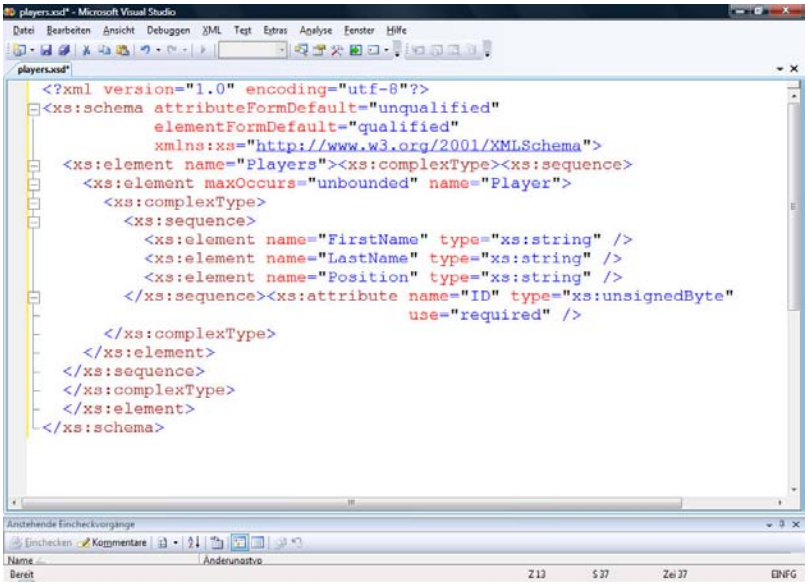


Abbildung 9.31
Erzeugte Schemadatei

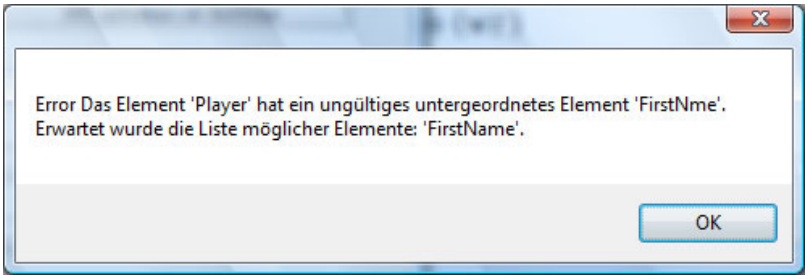
Listing 9.32 zeigt die entsprechende Routine.

```
Public Sub Validieren()
 Dim rsettings As New XmlReaderSettings
 rsettings.ValidationType = ValidationType.Schema

 'Fehlerhandler hinzufügen
 AddHandler rsettings.ValidationEventHandler, _
        AddressOf Fehlerbehandlung

 'Schema hinzufügen
 rsettings.Schemas.Add(Nothing, "players.xsd")

'Den Reader initialisieren
 Dim r As XmlReader = _
   XmlReader.Create("players.xml", rsettings)

'Dokument lesen
 Do While r.Read
 Loop
End Sub

Public Sub Fehlerbehandlung(ByVal sender As Object, _
   ByVal e As Schema.ValidationEventArgs)

    MessageBox.Show(e.Severity.ToString & _
        " " & e.Message)
End Sub
```

Listing 9.32
Validation von XML-Dokumenten

Sie sollten beim ersten Probieren keine Fehlermeldung erhalten, da Sie die xsd-Datei richtig erstellt haben und sie zu dem Dokument passt. Sobald Sie aber beispielsweise das Element *FirstName* in *FirstNme* umändern, hat das die Meldung aus Abbildung 9.32 zur Folge.

Abbildung 9.32 Fehlermeldung bei der Validierung



9.8.7 Transformation mit XSLT

Das Format Extensible Stylesheet Language dient zur Umwandlung einer XML-Datei in ein beliebiges anderes Format. Das Format ermöglicht die Kommunikation mit verschiedenen Anwendungstypen. Mit einer XSLT-Datei lassen sich XML-Dateien in verschiedene Formate bringen. So ist es zum Beispiel möglich, eine XML-Datei in eine HTML-Datei zu formatieren und auszugeben. In unserem Fall wurde eine XSLT-Datei bereits vorbereitet, die später für die Ausgabe der *players.xml* in eine statische HTML-Seite dienen soll:

Listing 9.33 xsl-Datei

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0">
<xsl:output method="html" encoding="iso-8859-1"
doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN" doctype-
system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>
<xsl:template match="/">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1"/>
<title>Special Teams</title>
<style type="text/css">
<xsl:comment>
.Stil1 {
 color: #FF0000;
 font-weight: bold;
}
</xsl:comment>
</style>
</head>
<body>
<p><strong>Liste aller Player </strong></p>
<xsl:for-each select="Players/Player">
  <table width="706" height="145" border="0">
    <tr>
```

Listing 9.33 (Forts.)
xsl-Datei

```
      <td width="172"><div align="left"
          class="Stil1">ID</div></td>
      <td width="524"><xsl:value-of select="@ID"/></td>
    </tr>
    <tr>
      <td><div align="left"><strong>VorName</strong></div></td>
      <td><xsl:value-of select="FirstName"/></td>
    </tr>
    <tr>
      <td><div align="left">
          <strong>Nachname</strong></div></td>
      <td><xsl:value-of select="LastName"/></td>
    </tr>
    <tr>
      <td><div align="left">
          <strong>Position</strong></div></td>
      <td><xsl:value-of select="Position"/></td>
    </tr>
  </table>

</xsl:for-each>
</body>
</html>

</xsl:template>
</xsl:stylesheet>
```

In der späteren HTML-Seite sollen alle Spieler formatiert ausgegeben werden. Die Datei wird von Visual Basic benötigt, um die formatierte Ausgabe vorzubereiten. Neben der XSL-Datei werden noch die XML-Datei sowie die neue Ausgabedatei benötigt.

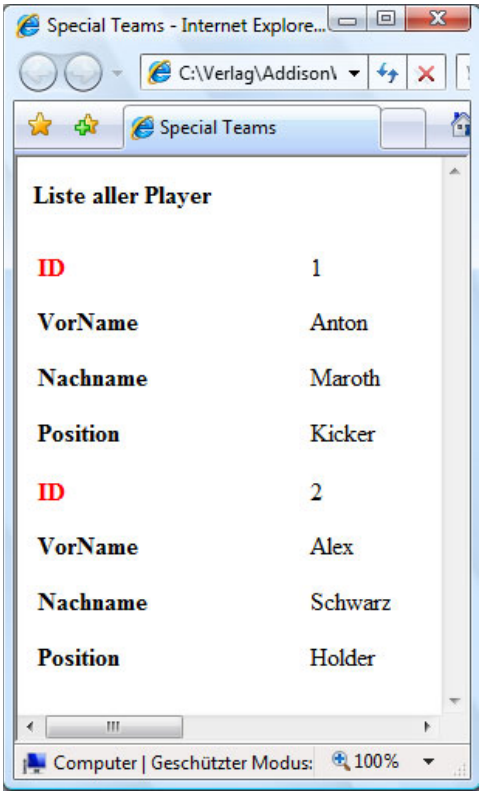
Danach kann die Ausgabe bereits erfolgen, wie Sie in Listing 9.34 sehen.

Listing 9.34
Umwandlung einer XML-Datei mit Hilfe von XSLT

```
Public Sub xslt()
  Dim xmlDatei As String = "Players.xml"
  Dim xslDatei As String = "Players.xsl"
  Dim ausgabeDatei As String = "ausgabe.html"

  Dim xslt As New Xsl.XslCompiledTransForm
  Try
    'XSL laden
    xslt.Load(xslDatei)

   'Umwandlung starten
    xslt.TransForm(xmlDatei, ausgabeDatei)
  Catch ex As Exception
    MessageBox.Show(ex.Message)
  End Try
End Sub
```

In der Abbildung 9.33 sehen Sie das erzeugte HTML-Dokument.

Abbildung 9.33
Ausgabe der mit XSLT erzeugten HTML-Seite



9.8.8 XML und das TableAdapter-Objekt

Das .NET Framework erlaubt es Ihnen, Daten direkt aus einem `TableAdapter` zu lesen oder direkt in ein `TableAdapter`-Objekt zu schreiben. Das hat gegenüber den konventionellen Möglichkeiten einen gewaltigen Performancevorteil. Im folgenden Beispiel wird zunächst ein `DataSet` mit einem Tabellenschema gefüllt, bevor neue, zufällig erzeugte Inhalte geschrieben werden. Das `TableAdapter`-Objekt liest schließlich den generierten Inhalt aus und schreibt diesen mit der Methode `WriteXML()` in ein XML-Dokument.

Listing 9.35
DataTable in ein XML-Dokument schreiben

```
Public Sub xmlschreiben()
  Dim tbl As DataTable
  Dim dr As DataRow
  Dim ds As New Dataset()

  'Zufällige Werte erzeugen
  Dim rnd As New Random()
  Dim rndWert As Integer

  ds.Tables.Add("ZufälligeDaten")
  ds.Tables("ZufälligeDaten").Columns.Add("ID")
  ds.Tables("ZufälligeDaten").Columns.Add("Wert")

    'Inhalte schreiben
```

```
  For i As Integer = 1 To 60
    Try
      rndWert = rnd.Next()
      dr = ds.Tables("ZufälligeDaten").NewRow()
      dr("ID") = i
      dr("Wert") = rndWert
      ds.Tables("ZufälligeDaten").Rows.Add(dr)

    Catch ex As Exception
      MessageBox.Show(ex.Message)
    End Try
  Next

  'DataTable lesen
  tbl = ds.Tables("ZufälligeDaten")
  Try
    'Daten über die Methode WriteXml schreiben
    tbl.WriteXml("Daten.xml")
  Catch ex As Exception
    MessageBox.Show(ex.Message)
  End Try
End Sub
```

Listing 9.35 (Forts.)
DataTable in ein XML-Dokument schreiben

Das erzeugte Dokument sehen Sie in Abbildung 9.34.

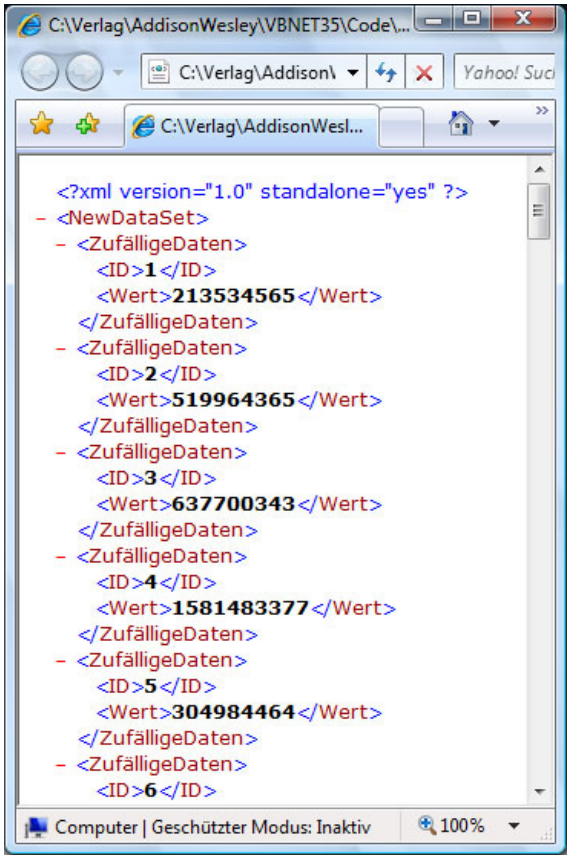


Abbildung 9.34
XML Daten – durch eine DataTable erzeugt

10

LINQ – Language Integrated Query

LINQ (Language Integrated Query) wurde mit dem .NET Framework 3.5 eingeführt. Mittels LINQ stehen in den Sprachen C# und VB neue Sprachelemente zur Verfügung, die auf eine einheitliche Art und Weise Zugriff auf unterschiedliche Arten von Daten ermöglichen. Dieses Kapitel soll eine kurze Einführung in die Bestandteile dieser Technologie geben.

10.1 Was ist LINQ?

LINQ bietet einen einheitlichen Zugriff auf Daten, egal ob diese in relationalen Datenbanken, in XML-Dateien oder in beliebigen Objektstrukturen vorliegen. Durch die Integration von Abfrageausdrücken in die .NET-Sprachen brauchen Sie jetzt für den Zugriff auf Daten nicht mehr unterschiedliche Abfrageausdrücke lernen, sondern haben mittels LINQ jetzt eine zentrale Abfragesprache für jegliche Art von Daten.

LINQ gliedert sich dabei in drei Teilbereiche:

- LINQ to Objects

  Mittels LINQ to Objects können Sie auf Daten, die innerhalb eines Objektmodells vorliegen, zugreifen und diese auch manipulieren.

- LINQ to ADO.NET

  LINQ to ADO.NET gliedert sich wiederum in zwei Unterbereiche:

  - LINQ to DataSet

  Mittels LINQ to DataSet haben Sie Zugriff auf Daten, die bereits in typisierten DataSets geladen sind. Diese Daten können natürlich auch manipuliert werden.

  - LINQ to SQL

  LINQ to SQL bietet Datenzugriff auf relationale Datenbanken, ohne die Abfragesprache SQL als Entwickler verwenden zu müssen. LINQ to SQL ist nur für den Microsoft SQL Server verfügbar.

- LINQ to XML

  Mit LINQ to XML können Sie auf Daten aus XML-Dateien zugreifen und diese auch manipulieren.

- LINQ to Entities

  Mit LINQ to Entities können Sie auf Daten aus relationalen Datenbanken zugreifen und diese bearbeiten. Dabei ist der Einsatz von SQL nicht nötig. Im Gegensatz zu LINQ to SQL steht LINQ to Entities auch für andere Datenbanken wie MS SQL Server zur Verfügung.

Diese Technologie bietet folgende Vorteile:

- Einheitliche API für den Datenzugriff

  Abfrageausdrücke sind in den Sprachen C# oder VB enthalten.

- Syntaxüberprüfung des Abfrageausdrucks durch den Compiler

  SQL-Ausdrücke wurden als Strings an die Datenbank übergeben, somit konnte der Compiler die Syntax der SQL-Ausdrücke nicht überprüfen. Fehlerhafte SQL-Syntax führte somit zu Laufzeitfehlern, jetzt ist die Abfragesprache Teil der Sprache, die der Compiler syntaktisch überprüfen kann.

- LINQ bietet absolute Typensicherheit

  Die Ergebnisse, die eine Datenbank aufgrund von SQL-Befehlen zurückliefert, sind untypisiert. Mittels LINQ werden diese Daten streng typisiert, um die relationale Welt mit der objektorientierten Welt zu verbinden.

So nun betrachten wir die einzelnen Bereiche von LINQ.

10.2 LINQ to Objects

LINQ to Objects ist der Teilbereich der LINQ-Technologie zum Zugriff auf Daten innerhalb eines Objektmodells. Mit LINQ to Objects können Sie auf sämtliche Objekte zugreifen, die das generische Interface `IEnumerable<Of T>` implementieren. Diese Objekte werden mit sogenannten Extensionsmethoden um entsprechende Funktionalitäten erweitert.

10.2.1 Erweiterungsmethoden

Bei Extensionsmethoden, oder zu Deutsch Erweiterungsmethoden, handelt es sich um einen Satz von Methoden, die bestehende Klassen erweitern, ohne dass die Originalklasse verändert wird oder von dieser abgeleitet wird. Die Erweiterungsmethoden für die Objekte, die das Interface `IEnumerable(Of T)` implementieren, liegen im Namespace `System.Linq` vor.

Abbildung 10.1 und Abbildung 10.2 zeigen die Verwendung eine generischen Liste, einmal mit der Einbindung des Namensraums, in dem die Erweiterungsmethoden definiert sind, und einmal ohne. In der ersten Abbildung sehen Sie, dass die Erweiterungsmethoden in der Vorschlagsliste mit einem anderen Symbol (zusätzlich zum Methodensymbol noch ein nach unten zeigender blauer Pfeil) gekennzeichnet sind.

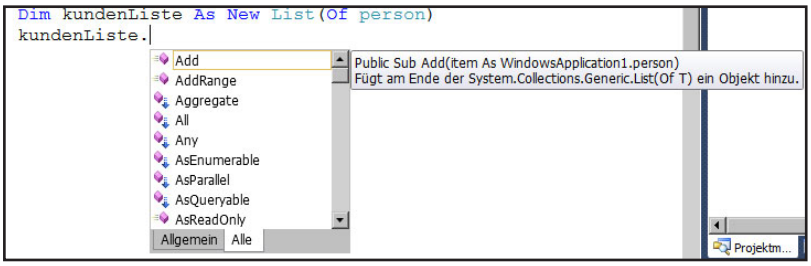


Abbildung 10.1
Verwendung einer generischen Liste mit Einbindung des Namespace `System.Linq`

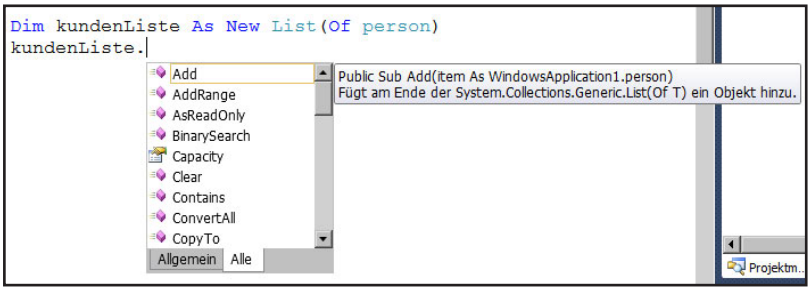


Abbildung 10.2
Verwendung einer generischen Liste ohne Einbindung des Namespace `System.Linq`

Tipp

Sie können auch für Typen, die nicht im .NET Framework definiert sind, Erweiterungsmethoden implementieren.

10.2.2 Standard-Query-Operatoren

Als Standard-Query-Operatoren bezeichnet man diejenigen in die Programmiersprachen neu eingeführten Operatoren, die Abfragefunktionalität beinhalten

In der nachfolgenden Tabelle 10.1 finden Sie einen Überblick über die wichtigsten Standard-Query-Operatoren.

Tabelle 10.1
Überblick über die wichtigsten Standard-Query-Operatoren

Standard Query Operator	Bedeutung
`Aggregate()`	Akkumuliert nach bestimmten Vorgaben bestimmte Werte einer generischen Liste
`Concat()`	Fasst zwei generische Listen zusammen
`Distinct()`	Gibt eine generische Liste mit eindeutigen Werten aus einer generischen Liste zurück
`ElementAt()`	Gibt ein Element an einer bestimmten Indexposition aus einer generischen Liste zurück
`First()`	Gibt das erste Element einer generischen Liste zurück
`GroupBy()`	Gruppiert eine generische Liste
`Join()`	Verknüpft zwei generische Listen aufgrund übereinstimmender Schlüsselfelder

Tabelle 10.1 (Forts.) Überblick über die wichtigsten Standard-Query-Operatoren

Standard Query Operator	Bedeutung
`Last()`	Gibt das letzte Element einer generischen Liste zurück
`Max()`	Gibt den maximalen Wert innerhalb einer generischen Liste zurück
`Min()`	Gibt den minimalen Wert innerhalb einer generischen Liste zurück
`OrderBy()`	Sortiert eine generische Liste auf der Basis eines anzugebenden Felds
`OrderByDescending()`	Sortiert eine generische Liste auf der Basis eines anzugebenden Felds in umgekehrter Reihenfolge
`Select()`	Selektiert Elemente aus einer generischen Liste
`Skip()`	Überspringt eine Anzahl von Elementen innerhalb einer generischen Liste
`SkipWhile()`	Überspringt Elemente mit bestimmten Bedingungen aus einer generischen Liste
`Sum()`	Summiert bestimmte Felder einer generischen Liste
`Union()`	Vereinigt zwei generische Listen des gleichen Typs
`Where()`	Filtert die Elemente einer generischen Liste aufgrund von bestimmten Kriterien

10.2.3 Beispielanwendung

Im Folgenden wollen wir mit einem kleinen Einführungsbeispiel die Mächtigkeit von LINQ to Objects demonstrieren.

Dazu benötigen wir eine Klasse `Kunde`, die lediglich zwei Eigenschaften `Name` und `Region` besitzt.

Listing 10.1 Listing der Kundenklasse

```
Public Class Kunde

    Private mname As String
    Public Property Name() As String
        Get
            Return mname
        End Get
        Set(ByVal value As String)
            mname = value
        End Set
    End Property

    Private mregion As String
    Public Property Region() As String
        Get
            Return mregion
        End Get
```

```
        Set(ByVal value As String)
            mregion = value
        End Set
    End Property

End Class
```

Listing 10.1 (Forts.)
Listing der Kundenklasse

Zur Vereinfachung des Codes füge ich in die Kundenklasse eine statische Methode `GetKunden()` hinzu, die mir eine generische Liste von Objekten des Typs `Kunde` zurückgibt. Diese Methode ist in Listing 10.2 dargestellt.

```
Public Shared Function GetKunden() As List(Of Kunde)
  Dim kListe As New List(Of Kunde)
  kListe.Add(New Kunde With { _
    .Name = "Münchner", .Region = "BY"})
  kListe.Add(New Kunde With { _
    .Name =   "Augsburger", .Region = "BY"})
  kListe.Add(New Kunde With { _
    .Name = "Regensburger", .Region = "BY"})
  kListe.Add(New Kunde With { _
    .Name = "Stuttgarter", .Region = "BW"})
  kListe.Add(New Kunde With { _
    .Name = "Berliner", .Region = "BER"})
  kListe.Add(New Kunde With { _
    .Name = "Frankfurter", .Region = "HES"})
  kListe.Add(New Kunde With { _
    .Name = "Hamburger", .Region = "HH"})
  kListe.Add(New Kunde With { _
    .Name = "Bremer", .Region = "BRE"})
  kListe.Add(New Kunde With { _
    .Name = "Kölner", .Region = "NRW"})
  kListe.Add(New Kunde With { _
    .Name = "Düsseldorfer", .Region = "NRW"})
  kListe.Add(New Kunde With { _
    .Name = "Kieler", .Region = "NIE"})

    Return kListe
End Function
```

Listing 10.2
Methode GetKunden()

Hier erstellen wir eine generische Liste, die Objekte unseres Typs `Kunde` aufnehmen kann. Im Anschluss daran generieren wir mit Hilfe der vereinfachten Objektinitialisierung Kunden in verschiedenen Bundesländern und fügen diese der Liste, die wir am Ende der Funktion zurückgeben, hinzu.

Auf einem Formular wollen wir jetzt in einer `ComboBox` alle verfügbaren Bundesländer, natürlich ohne Duplikate, anzeigen.

```
Private Sub frmLINQtoObjects_Load(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load

   Dim kundenListe As List(Of Kunde) = Kunde.GetKunden()
   Dim laenderListe = From k In kundenListe Order By k.Region
       Select k.Region Distinct

   For Each reg As String In laenderListe
     cboBundesland.Items.Add(reg)
   Next
End Sub
```

Listing 10.3
Befüllen der Drop-down-Liste mit Bundesländern

Hier sehen Sie zum ersten Mal die Spracherweiterungen für LINQ. Beim Laden des Formulars wird zuerst eine Kundenliste mit der statischen `GetKunden()`-Methode initialisiert. Danach erstellen wir eine neue Liste mit Bundesländern auf der Basis eines LINQ-Ausdrucks, den wir uns nun genauer betrachten wollen:

```
From k In kundenListe Order By k.Region
   Select k.Region Distinct
```

Auch wenn Sie dieser Ausdruck wohl ein bisschen an SQL erinnert, ist es auf den ersten Blick wohl etwas verwirrend, dass das `Select`-Schlüsselwort nicht am Anfang des Ausdrucks steht.

Zerlegen wir den Ausdruck in die folgenden Teile:

- From k in kundenListe

  Mittels `From` lesen wir ein Objekt `k` aus der `kundenListe`. Da es sich bei der `kundenListe` um eine generische Liste vom Typ `Kunde` handelt, wird die Objektvariable `k` automatisch als Objekt vom Typ `Kunde` erkannt. Sie haben auch für die Variable `k` vollständige IntelliSense-Unterstützung, da VB seit der Version 9 implizite Typisierung unterstützt.

- Order By k.Region

  Mittels `Order` wird die Liste, in diesem Fall nach dem Attribut Region, sortiert.

- Select k.Region Distinct

  Mit `Select` wird definiert, welche Attribute aus dem Objekt `Kunde` herausgelesen werden sollen. In diesem Fall nur die Region. Das Schlüsselwort `Distinct` besagt noch, dass jeder Wert (jede Region) nur ein einziges Mal zurückgegeben wird. Somit wird ein neuer anonymer Typ gebildet, der nur aus einem Attribut `Region` besteht.

Abbildung 10.3 zeigt die befüllte ComboBox.

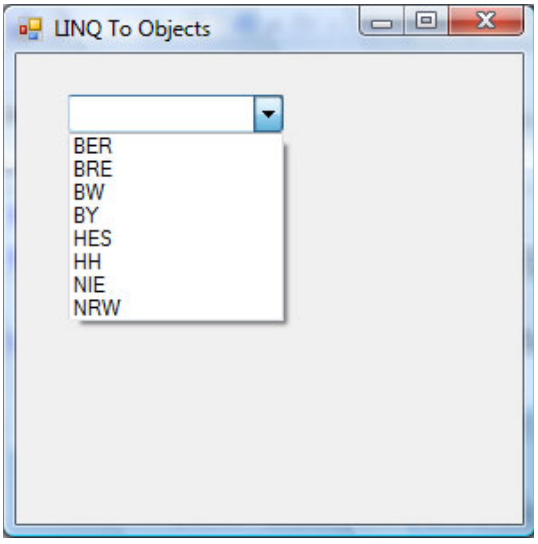


Abbildung 10.3
Liste aller enthaltenen Bundesländer

Im nächsten Schritt wollen wir bei einer Selektion eines Listeneintrags die Kunden aus diesem Bundesland in einer `DataGridView` betrachten.

Listing 10.4 zeigt den Code zum Befüllen des `DataGridView`-Steuerelements mit den Kundendaten.

Listing 10.4
Code zum Befüllen der GridView mit Kunden aus einem gewählten Bundesland

```
Private Sub cboBundesland_SelectedIndexChanged _
  (ByVal sender As Object, ByVal e As System.EventArgs) _
   Handles cboBundesland.SelectedIndexChanged

   If cboBundesland.SelectedIndex > -1 Then
     Dim kundenListe As List(Of Kunde) = Kunde.GetKunden()

     Dim liste As New List(Of Kunde)

     For Each ku As Kunde In
         From k In kundenListe _
         Where k.Region = cboBundesland.Text _
         Select k

         liste.Add(ku)
     Next

     dgvKunden.DataSource = liste

  End If
End Sub
```

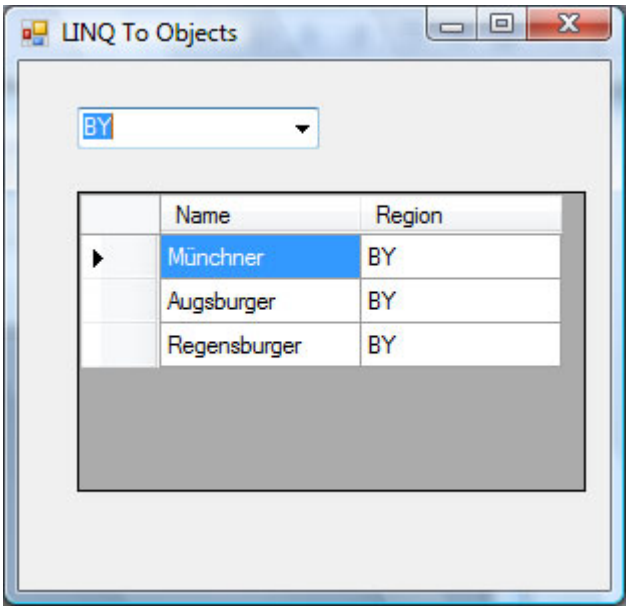
Auch hier wird wiederum mit der statischen `GetKunden()`-Methode die `kundenListe` befüllt. Anschließend wird eine generische Liste von Kunden definiert. Mittels eines LINQ-Ausdrucks holen wir uns die Kunden aus einem bestimmten Bundesland aus der `kundenListe` heraus und befüllen damit die generische Liste. Die Datenquelle der `DataGridView` wird danach auf die befüllte `liste` gesetzt.

Im Vergleich zum vorigen Beispiel, denke ich, liest sich das jetzt schon wesentlich klarer. Wir holen uns wiederum Kundenobjekte aus der Liste und filtern mit dem Schlüsselwort `Where` nach dem gewählten Bundesland. Mit `Select k` wird das gesamte Kunden-Objekt selektiert, also alle Attribute dieses Objekts.

Abbildung 10.4 zeigt die Ausgabe der gerade erstellten Seite im Browser.

In diesem Abschnitt wollte ich Ihnen einen kurzen Einblick in *LINQ to Objects* bieten. Die Möglichkeiten, die sich hinter dieser Technologie verbergen, wurden aber nur gestreift.

Abbildung 10.4
Befüllte DataGridView in der Beispielanwendung



10.3 LINQ to ADO.NET

Nun wollen wir uns um den Zugriff auf Daten in relationalen Datenbanken kümmern. Dabei will ich Ihnen vor allem *LINQ to SQL* vorstellen. *LINQ to DataSet* funktioniert sehr ähnlich zu *LINQ to Objects*, nur dass Sie dazu Ihre `DataTables` mit herkömmlichen ADO.NET-Bordmitteln bereits befüllt haben müssen und anstatt auf generische Listen letztendlich auf `DataTables` innerhalb eines typisierten `DataSet` zugreifen. `LINQ to ADO.NET` ist somit der Teil von LINQ, der es ermöglicht, auf Daten in relationalen Datenbanken zuzugreifen. Allerdings will ich gleich noch eine Bemerkung vorwegschicken: Mit dem Service Pack 1 des .NET Framework 3.5 wurde das *Entity Framework* eingeführt, das für Neuentwicklungen *LINQ to SQL* vorzuziehen ist.

10.3.1 LINQ to SQL

LINQ to SQL funktioniert ausschließlich mit Microsoft SQL Server-Datenbanken. Die Funktionalität von *LINQ to SQL* steckt dabei in der Bibliothek *System.Data.LINQ.dll*. Diese Bibliothek muss in jedem Projekt referenziert werden, in dem *LINQ to SQL* verwendet werden soll.

Mittels *LINQ to SQL* wird die relationale Datenbankstruktur in ein Objektmodell überführt. Dies geschieht durch ein objektrelationales Mapping. Man kann *LINQ to SQL* also durchaus als einen OR-Mapper bezeichnen. Das bedeutet, dass für die Tabellen aus der Datenbank innerhalb der Anwendung streng typisierte Klassen zur Verfügung stehen, die ein Mapping auf die Tabelle besitzen. Diese Klassen werden auch als *Entitätsklassen* bezeichnet.

Dabei kann das Mapping auf drei verschiedene Wege durchgeführt werden:

- Manuell

  Hierbei wird das Objektmodell manuell angelegt und über Attribute werden die Propertys der Klasse auf eine Tabelle in einer relationalen Datenbank gemappt. Hierfür stehen zwei unterschiedliche Attribute zur Verfügung – `Table` und `Column`.

- Designer

  Innerhalb von Visual Studio 2010 können Sie mittels eines Designers das Objektmodell automatisch generieren lassen. Dies ist wohl auch der am häufigsten gewählte Weg und wird deshalb im folgenden Abschnitt genauer erläutert.

- SqlMetal

  Bei `SqlMetal` handelt es sich um ein Befehlszeilentool zur Generierung des Objektmodells. `SqlMetal` ist dabei sehr exakt parametrisierbar, um genau die gewünschten Teile des Datenbankmodells in das neu zu erstellende Objektmodell zu überführen.

Bevor wir uns die Generierung des Objektmodells mittels des Visual Studio Designer genauer betrachten, wollen wir noch kurz das `DataContext`-Objekt vorstellen.

10.3.2 DataContext

Eine der wichtigsten Klassen der Bibliothek *System.Data.Linq.dll* ist die Klasse `DataContext`. Diese Klasse kommuniziert zwischen dem Objektmodell und der relationalen Datenbank. Dabei erfolgt die Kommunikation in beide Richtungen. Die `DataContext`-Klasse ist verantwortlich für das Befüllen des Objektmodells aus der Datenbank sowie für das Persistieren der gemachten Änderungen in den Daten des Objektmodells in die Datenbank. Dabei werden auch alle LINQ-Ausdrücke auf Basis der Attribute innerhalb der Entitätsklassen in SQL umgewandelt.

Das `DataContext`-Objekt benötigt auch eine gültige Verbindungszeichenfolge zur Datenbank. Diese kann entweder durch Übergabe eines gültigen *Connectionstrings* oder eines ADO.NET-*Connectionobjekts* an den Konstruktor erfolgen.

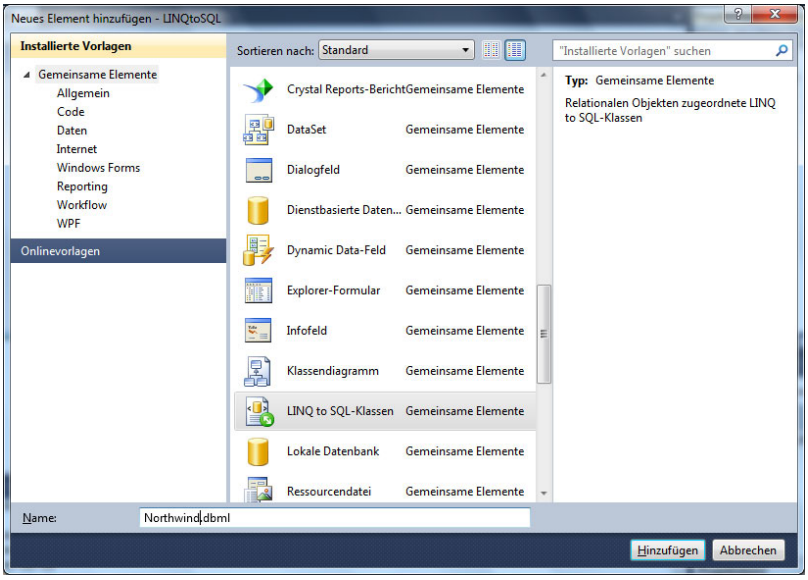
Der `DataContext` wird bei der Verwendung des im folgenden Kapitel beschriebenen Designers automatisch generiert.

10.3.3 LINQ to SQL Klassendesigner

Wenn Sie innerhalb von Visual Studio 2010 das OR-Mapping mittels des Designers anlegen wollen, gehen Sie wie folgt vor.

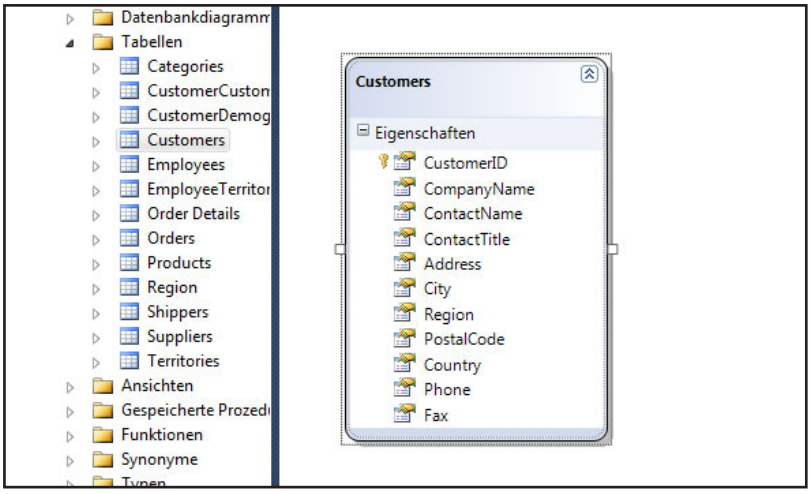
1. Fügen Sie über das Kontextmenü des Projekts ein neues Element zu Ihrem Projekt hinzu. Wählen Sie dazu aus dem nachfolgenden Dialog *LINQ to SQL Klassen* aus, wie in Abbildung 10.5 dargestellt, und vergeben Sie den Namen *Northwind.dbml*.

Abbildung 10.5
Auswahl des LINQ zu SQL-Klassen-Designer



2. Anschließend können Sie sich über den Server-Explorer per Drag&Drop beliebige Datenbankobjekte in die Designerübersicht ziehen. In unserem kleinen Beispiel ziehen wir nur die Customers-Tabelle aus der Nordwind-Datenbank in den Designer. Das Ergebnis sollte so aussehen, wie in Abbildung 10.6 dargestellt.
3. Im Projektmappenexplorer sollten Sie danach die gewünschten Dateien zur Verfügung haben, siehe Abbildung 10.7.

Abbildung 10.6
Designeransicht



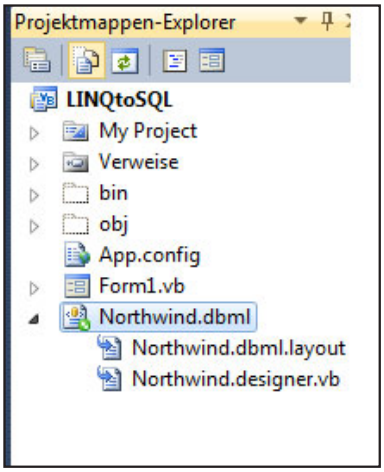


Abbildung 10.7
Ausschnitt aus dem Projektmappenexplorer

In der Datei *Northwind.designer.vb* befinden sich jetzt die `DataContext`-Klasse sowie eine Entitätsklasse für die Tabelle `Customers`.

Die Verbindungszeichenfolge für die `DataContext`-Klasse, die in diesem Beispiel `NordwindDataContext` heißt, befindet sich in der *app.config* (Listing 10.5).

Listing 10.5
ConnectionString in der app.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
  </configSections>
  <connectionStrings>
    <add name= "LINQtoSQL.My.MySettings
              .NorthwindConnectionString"
         connectionString="Data Source=localhost\SqlExpress;
                           Initial Catalog=Northwind;
                           Integrated Security=True"
         providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Listing 10.6 zeigt einen Ausschnitt aus der `Customers`-Klasse, die Entitätsklasse für die Tabelle *Customers*. Aus Platzgründen werden nur drei Eigenschaften angezeigt, für die anderen ist der Code identisch.

Listing 10.6
Code der Entitätsklasse Customers

```
<Table(Name:="dbo.Customers")>  _
Partial Public Class Customer
  Implements _
        System.ComponentModel.INotifyPropertyChanging, _
        System.ComponentModel.INotifyPropertyChanged

  Private Shared emptyChangingEventArgs _
          As PropertyChangingEventArgs = _
          New PropertyChangingEventArgs(String.Empty)

    Private _CustomerID As String
    Private _CompanyName As String
    Private _ContactName As String
```

Listing 10.6 (Forts.) Code der Entitätsklasse Customers

```
#Region "Extensibility Method Definitions"
  Partial Private Sub OnLoaded()
  End Sub
  Partial Private Sub OnValidate _
    (action As System.Data.Linq.ChangeAction)
  End Sub
  Partial Private Sub OnCreated()
  End Sub
  Partial Private Sub OnCustomerIDChanging(value As String)
  End Sub
  Partial Private Sub OnCustomerIDChanged()
  End Sub
  Partial Private Sub OnCompanyNameChanging(value As String)
  End Sub
  Partial Private Sub OnCompanyNameChanged()
  End Sub
  Partial Private Sub OnContactNameChanging(value As String)
  End Sub
  Partial Private Sub OnContactNameChanged()
  End Sub
#End Region

  Public Sub New()
    MyBase.New
      OnCreated
    End Sub

 <Column(Storage:="_CustomerID", DbType:="NChar(5) NOT NULL", _
   CanBeNull:=False, IsPrimaryKey:=True)>  _
 Public Property CustomerID() As String
   Get
     Return Me._CustomerID
   End Get
   Set
     If (String.Equals(Me._CustomerID, value) = False) Then
       Me.OnCustomerIDChanging(value)
       Me.SendPropertyChanging
       Me._CustomerID = value
       Me.SendPropertyChanged("CustomerID")
       Me.OnCustomerIDChanged
     End If
   End Set
 End Property

 <Column(Storage:="_CompanyName",
   DbType:="NVarChar(40) NOT NULL", CanBeNull:=False)>  _
 Public Property CompanyName() As String
   Get
     Return Me._CompanyName
   End Get
   Set
     If (String.Equals (Me._CompanyName, value) = False) Then
       Me.OnCompanyNameChanging(value)
       Me.SendPropertyChanging
       Me._CompanyName = value
```

Listing 10.6 (Forts.) Code der Entitätsklasse Customers

```
        Me.SendPropertyChanged("CompanyName")
        Me.OnCompanyNameChanged
      End If
    End Set
  End Property

  <Column(Storage:="_ContactName", DbType:="NVarChar(30)")>  _
  Public Property ContactName() As String
    Get
      Return Me._ContactName
    End Get
    Set
      If (String.Equals(Me._ContactName, value) = False) Then
        Me.OnContactNameChanging(value)
        Me.SendPropertyChanging
        Me._ContactName = value
        Me.SendPropertyChanged("ContactName")
        Me.OnContactNameChanged
      End If
    End Set
  End Property
   Public Event PropertyChanging As _
     PropertyChangingEventHandler Implements _
     System.ComponentModel.INotifyPropertyChanging. _
                           PropertyChanging

   Public Event PropertyChanged As _
     PropertyChangedEventHandler Implements _
     System.ComponentModel. _
     INotifyPropertyChanged.PropertyChanged

   Protected Overridable Sub SendPropertyChanging()
     If ((Me.PropertyChangingEvent Is Nothing) = False) Then
       RaiseEvent PropertyChanging(Me, emptyChangingEventArgs)
     End If
   End Sub

   Protected Overridable Sub SendPropertyChanged _
     (ByVal propertyName As [String])
     If ((Me.PropertyChangedEvent Is Nothing) = False) Then
       RaiseEvent PropertyChanged(Me, _
            New PropertyChangedEventArgs(propertyName))
     End If
   End Sub
End Class
```

Auch wenn diese Klasse automatisch erstellt wurde, sollten wir zumindest einen Überblick bekommen, was in dieser Klasse alles drin steckt.

```
<Table(Name:="dbo.Customers")> _
Partial Public Class Customer
```

Das ist die Definition der Klasse mit dem zugehörigen `Table`-Attribut, das die Verknüpfung zwischen der Objektklasse und der Tabelle definiert.

```
Implements _
 System.ComponentModel.INotifyPropertyChanging, _
 System.ComponentModel.INotifyPropertyChanged
```

Anschließend erfolgt die Implementierung von zwei Interfaces, damit Änderungen an Daten entsprechende `Changed`- und `Changing`-Events auslösen können und Änderungen an den Daten auch im Objektmodell übernommen werden.

Nach der Definition von speziellen `EventArgs` und den privaten Feldern erfolgt die Definition von partiellen Methoden und schließlich der Konstruktor der Klasse.

Danach folgen die Propertys für die Attribute, die ein Mapping auf die einzelnen Tabellenfelder darstellen und im `Set`-Zweig die entsprechenden partiellen Methoden aufrufen.

Mit dem Attribut `Column` wird das Mapping auf das entsprechende Datenbankfeld durchgeführt sowie bestimmte Definitionen werden aus der Datenbank (Datentyp, Primärschlüssel etc.) bereitgestellt.

Im `Get`-Zweig wird der Wert der internen Feldvariablen zurückgegeben und im `Set`-Zweig wird bei einer Änderung des Werts dieser an die interne Feldvariable übergeben und entsprechende Events werden über partielle Methoden aufgerufen.

Listing 10.7 zeigt nochmals exemplarisch die Eigenschaft *CustomerID*.

Listing 10.7 Property CustomerID

```
<Column(Storage:="_CustomerID", DbType:="NChar(5) NOT NULL", _
 CanBeNull:=False, IsPrimaryKey:=True)>  _
Public Property CustomerID() As String
  Get
    Return Me._CustomerID
  End Get
  Set
    If (String.Equals(Me._CustomerID, value) = False) Then
        Me.OnCustomerIDChanging(value)
        Me.SendPropertyChanging
        Me._CustomerID = value
        Me.SendPropertyChanged("CustomerID")
        Me.OnCustomerIDChanged
    End If
  End Set
End Property
```

Am Ende werden dann noch die beiden Methoden zum Auslösen der `Changed`- und `Changing`-Events überschrieben.

Nachdem nun der Designer das OR-Mapping erstellt hat, wollen wir im nächsten Schritt die Daten auch in einer Anwendung darstellen.

10.3.4 Datenbindung mittels Designerklassen

Wir wollen auf einer neuen Seite alle amerikanischen Kunden aus der `Customers`-Tabelle in einer `GDataGridView` anzeigen. Dazu implementieren wir im `Form1_Load`-Ereignis den Code aus Listing 10.8.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load

   Dim context As New NordwindDataContext
   DataGridView1.DataSource = _
        From c In context.Customers _
        Where c.Country = "USA" _
        Select c.CustomerID, c.CompanyName, _
               c.City, c.Country, c.Phone
End Sub
```

Listing 10.8
Anzeigen der Kunden in einer GridView mittels LINQ to SQL

Beim Laden des Formulars instanziieren wir ein neues `NordwindDataContext`-Objekt. Die Verbindungszeichenfolge wird dabei aus der *app.config* gelesen. Danach setzen wir die Datenquelle des `DataGridView`-Steuerelements mittels eines LINQ-Ausdrucks.

```
From c In context.Customers _
   Where c.Country = "USA" _
   Select c.CustomerID, c.CompanyName, c.City, _
          c.Country, c.Phone
```

Über die Objektvariable `c` lesen wir aus der Entitätsklasse `Customers`, die als Property des zugehörigen `NordwindDataContext` implementiert wurde, alle Kunden aus, die als Land den Eintrag USA haben (`Where`-Schlüsselwort). In diesem Fall wollen wir nicht alle Spalten zurückgeben, sondern nur diejenigen, die wir dann in der `Select`-Anweisung definieren.

Abbildung 10.8 stellt die Ausgabe der Seite dar.

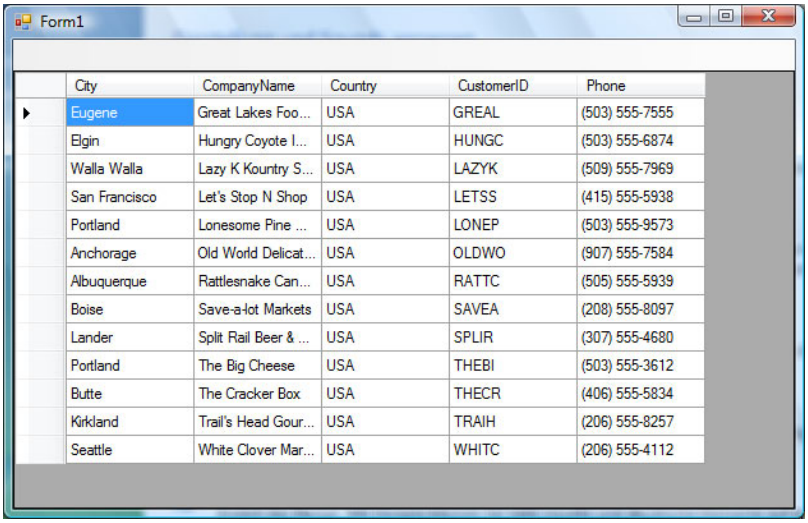


Abbildung 10.8
Kundenausgabe

10.3.5 Datenmanipulation

Nun wollen wir uns noch betrachten, wie groß der Aufwand ist, Daten zu ändern und in die Datenquelle zu persistieren.

Dazu füge ich dem Formular noch ein *Datei*-Menü mit den Unterpunkten *Speichern* und *Beenden* hinzu.

Als Zweites müssen wir den `DataContext` als private Variable für die gesamte Form definieren.

```
Private context As NordwindDataContext = _
        New NordwindDataContext
```

So, nun fehlt uns nur noch der Programmcode für das Speichern. Dieser besteht allein aus dem Aufruf der `SubmitChanges()`-Methode des `context`-Objekts.

```
context.SubmitChanges()
```

Wenn Sie die Anwendung jetzt testen, werden Sie feststellen, dass Sie in der `DataGridView` überhaupt keine Daten ändern können. Dies liegt an unserem LINQ-Ausdruck:

```
From c In context.Customers _
   Where c.Country = "USA" _
   Select c.CustomerID, c.CompanyName, c.City, _
          c.Country, c.Phone
```

Dieser Ausdruck gibt einen anonymen Typen zurück und für diesen anonymen Typen besteht keine Möglichkeit, Funktionalität zu implementieren. Das bedeutet, es gibt auch keine Funktionalität zum Speichern. Wenn Sie jedoch den Ausdruck so abändern, dass ein `Customer`-Objekt zurückgegeben wird, wird das Editieren funktionieren und die Daten werden auch erfolgreich in die Datenbank geschrieben.

```
From c In context.Customers _
   Where c.Country = "USA" _
   Select c
```

Listing 10.9 zeigt den gesamten Programmcode des Formulars.

Listing 10.9 Programmcode der gesamten Anwendung

```
Public Class Form1
  Private context As NordwindDataContext = _
    New NordwindDataContext

  Private Sub Form1_Load(ByVal sender As System.Object, _
     ByVal e As System.EventArgs) Handles MyBase.Load

     DataGridView1.DataSource = _
        From c In context.Customers _
        Where c.Country = "USA" _
        Select c
  End Sub
```

```
  Private Sub BeendenToolStripMenuItem_Click _
    (ByVal sender As System.Object, _
     ByVal e As System.EventArgs) _
     Handles BeendenToolStripMenuItem.Click

     Me.Close()
  End Sub

  Private Sub SpeichernToolStripMenuItem_Click _
    (ByVal sender As System.Object, _
     ByVal e As System.EventArgs) _
     Handles SpeichernToolStripMenuItem.Click

     context.SubmitChanges()
  End Sub
End Class
```

Listing 10.9 (Forts.)
Programmcode der gesamten Anwendung

10.4 LINQ to XML

LINQ to XML bietet eine API für den Zugriff auf Daten in XML-Dokumenten.

Damit können Sie XML-Dokumente anlegen, auslesen, abfragen und Änderungen an dem Dokument durchführen.

Um *LINQ to XML* nutzen zu können, benötigen Sie eine Referenz auf die Bibliothek *System.Xml.Linq.dll*.

Die wichtigsten Objekte in dieser Bibliothek sind:

- XDocument

  Referenziert auf ein XML-Dokument
- XAttribute

  Referenziert auf ein XML-Attribut
- XElement

  Referenziert auf ein XElement

Auch hierzu will ich ein kleines Beispiel zeigen. Legen Sie dazu bitte ein neues Projekt an und vergessen Sie nicht, den Verweis auf die Assembly `System.Xml.Linq` zu setzen.

In der `Form1_Load()`-Methode will ich mit den gerade aufgelisteten Objekten eine XML-Datei erzeugen.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load

   Dim doc As New XDocument( _
     New XDeclaration("1.0", Nothing, Nothing), _
     New XElement("Customers", _
       New XElement("Customer", _
        New XAttribute("CustomerId", "ALFKI"), _
```

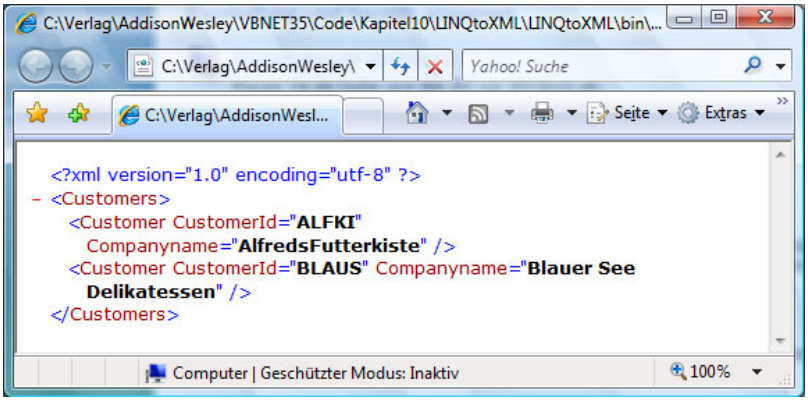
Listing 10.10
Erzeugen einer XML-Datei

Listing 10.10 (Forts.)
Erzeugen einer XML-Datei

```
            New XAttribute("Companyname", "AlfredsFutterkiste")), _
          New XElement("Customer", _
            New XAttribute("CustomerId", "BLAUS"), _
            New XAttribute("Companyname", _
                           "Blauer See Delikatessen"))))

  doc.Save("customers.xml")
End Sub
```

In Abbildung 10.9 sehen Sie die gerade erzeugte Datei.

Abbildung 10.9
Erzeugte customers.xml-Datei



Im zweiten Schritt wollen wir mit einer LINQ-Abfrage Daten aus dieser Datei auslesen. Dazu füge ich dem Formular einen `Button` und eine `ListBox` hinzu.

Listing 10.11
Auslesen von Daten aus der XML-Datei

```
Private Sub btnLaden_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnLaden.Click

   Dim d As XDocument = XDocument.Load("customers.xml")
   Dim kundenListe = From c In _
       d.Descendants("Customers").Descendants() _
       Where (c.Attribute("Companyname").Value. _
              StartsWith("A")) _
       Select New With {.Companyname = _
           c.Attribute("Companyname").Value, _
         .CustomerId = c.Attribute("CustomerId").Value}
        For Each item In kundenListe
            ListBox1.Items.Add(item.CustomerId)
        Next
End Sub
```

Wie Sie in Listing 10.11 sehen, erzeugen wir mit der statischen Methode `Load()` ein `XDocument` aus der zuvor erzeugten Datei. Anschließend erzeugen wir uns mit einem LINQ-Ausdruck eine Liste eines anonymen Typs, bestehend aus den Eigenschaften *Companyname* und *CustomerId*. Anschließend durchlaufen wir die Liste und befüllen die `ListBox` im Formular. Abbildung 10.10 zeigt die Liste mit dem Kunden *ALFKI*, dem einzigen Kunden, dessen Firmenname mit einem *A* beginnt.

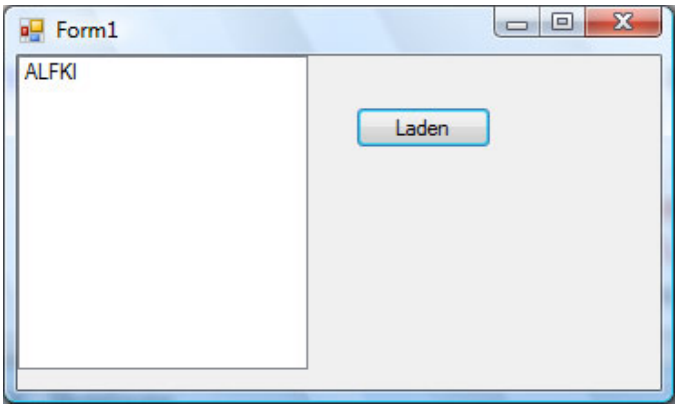


Abbildung 10.10
Ausgabe des XML-LINQ-Ausdrucks

10.5 LINQ-Pad

Unter *http://www.linqpad.net/* steht das Tool *LINQPad* zum Download bereit. Dieses Tool finde ich hervorragend geeignet, um sich in LINQ einzuarbeiten.

Sie können darin LINQ-Ausdrücke eintippen und sofort das Ergebnis aus der Datenbank sowie das erzeugte SQL-Statement betrachten. Außerdem steht Ihnen ein Tutorial mit vielen vordefinierten LINQ-Ausdrücken zur Verfügung.

Abbildung 10.11 zeigt die Auswertung eines LINQ-Ausdrucks und der entsprechenden Ergebnisanzeige, während in Abbildung 10.12 das erzeugte SQL-Statement dargestellt wird.

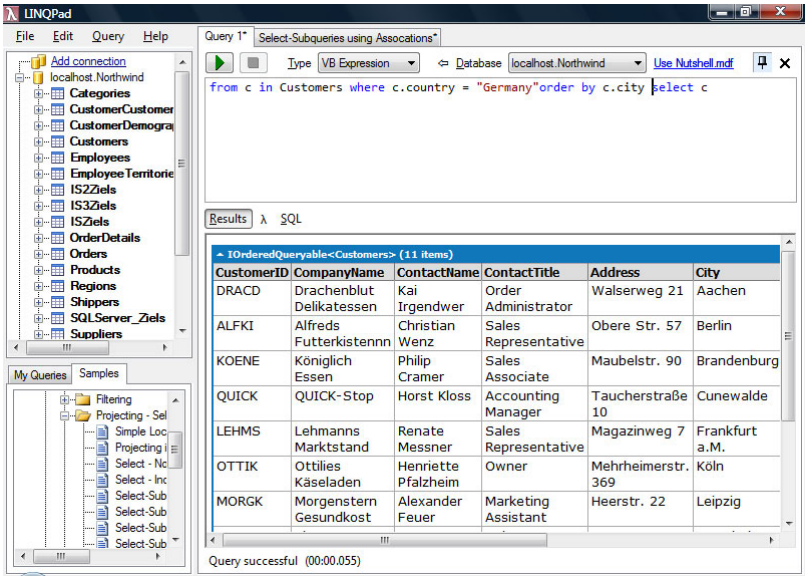
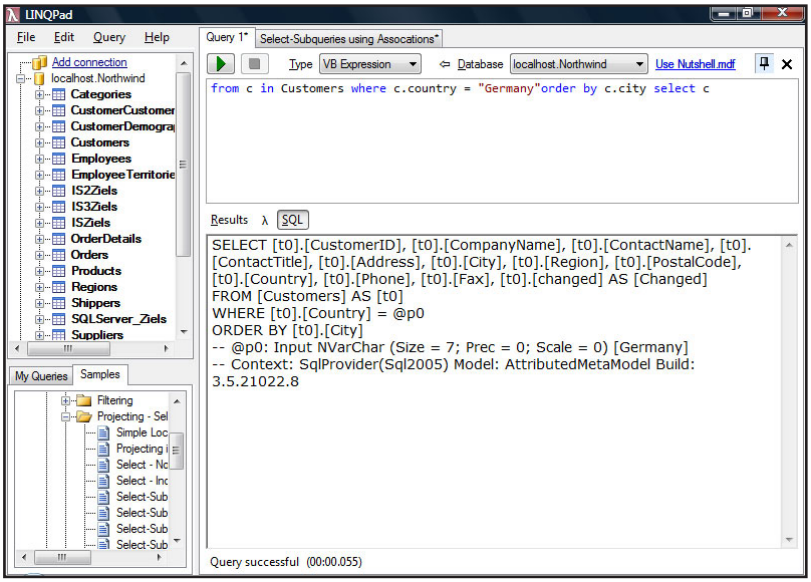


Abbildung 10.11
LINQ-Ausdruck mit Ergebnisanzeige

Abbildung 10.12
LINQ-Ausdruck mit Anzeige des erzeugten SQL



10.6 LINQ to Entities – Entity Framework

Mit dem Service Pack 1 des .NET Framework 3.5 wurde das *Entity Framework* mit *LINQ to Entities* eingeführt. Im Gegensatz zu *LINQ to SQL* unterstützt dieses Verfahren unterschiedliche Datenbanken, also nicht nur Microsoft SQL Server. Auch die Abbildung von sogenannten m:n-Beziehungen ist möglich (*LINQ to SQL* war auf 1:n-Beziehungen beschränkt). Das sind die offensichtlichsten Änderungen für den Entwickler, der sonst in der Art und Weise, wie er beide Verfahren einsetzt, kaum Unterschiede sieht.

Doch unter der Motorhaube sind beide Technologien schon ziemlich unterschiedlich. Das Entity Framework basiert auf einem Entitätsdatenmodell (*Entity Data Model*), das die Daten vom tatsächlichen Speichermodell komplett abstrahieren kann. Es handelt sich dabei also um einen ORM (objektrelationalen Mapper) wie zum Beispiel auch *NHibernate*.

Das Entity Data Model besteht aus drei Schichten:

- Konzeptionelle Schicht

  Modell, wie der Benutzer die Daten sieht und verwendet
- Speichermodell

  Modell, wie die Daten tatsächlich gespeichert werden
- Mapping-Modell

  Modell, wie das Speichermodell und das konzeptionelle Modell miteinander verbunden sind

Das Modell wird als XML-Struktur gespeichert.

Für die Verwendung von *LINQ to Entities* innerhalb von .NET gegenüber dem in unterschiedlichen Technologien bewährten NHibernate spricht natürlich die hervorragende Unterstützung innerhalb von Visual Studio während der Entwicklungszeit und die Integration von LINQ in der gewünschten Programmiersprache.

Wie auch bei *LINQ to SQL* gibt es drei Möglichkeiten, das Modell zu erstellen:

- Mit dem Befehlszeilentool *EDMGen*
- Mit dem ADO.NET Entity Data Modell-Assistenten in Visual Studio 2010
- Im Framework 4.0 gibt es außerdem die Möglichkeit, das Modell in Visual Studio zu erstellen und daraus eine Datenbank generieren zu lassen.

Betrachten wir im Folgenden den Visual Studio-Assistenten etwas genauer.

Fügen Sie zu Ihrem Projekt ein neues Element hinzu. Wählen Sie dabei die Vorlage *ADO.NET Entity Data Model* wie in Abbildung 10.13 dargestellt und als Name *Northwind.edmx*.

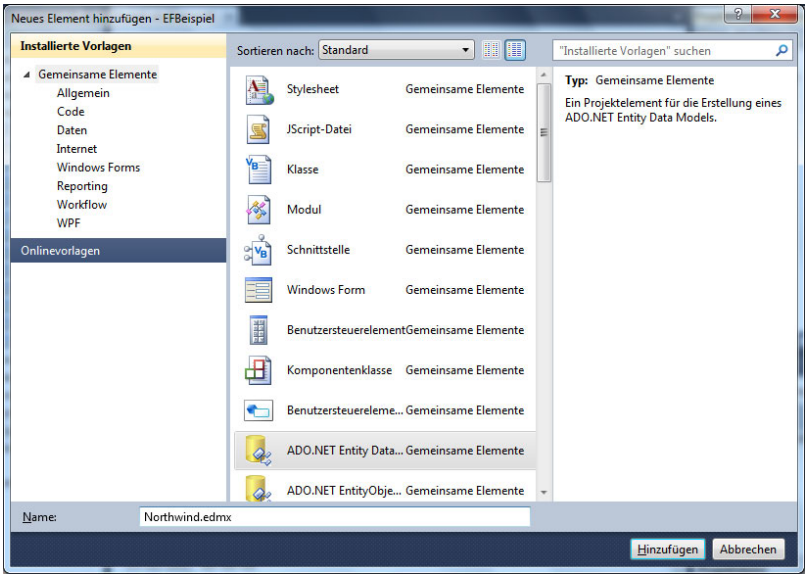


Abbildung 10.13 Auswahl des ADO.NET Entity Data Model-Assistenten

Im folgenden Dialog wählen Sie den Eintrag *Aus Datenbank generieren* aus und anschließend als *Connection* die in diesem Projekt bereits verwendete Verbindung zur Nordwinddatenbank (siehe Abbildung 10.14).

Jetzt wird die Datenbankstruktur ausgelesen und Sie können auswählen, welche Datenbankobjekte Sie in Ihrem Entitätsdatenmodell haben wollen.

Wählen Sie dazu nur die Tabelle *Customers* aus, wie in Abbildung 10.15 dargestellt.

Abbildung 10.14
Auswahl der Datenverbindung

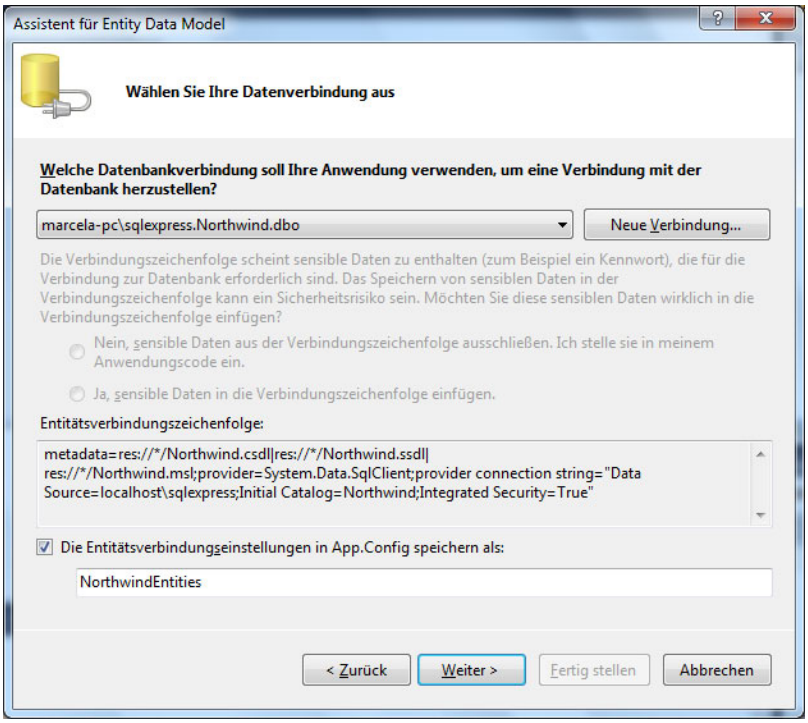
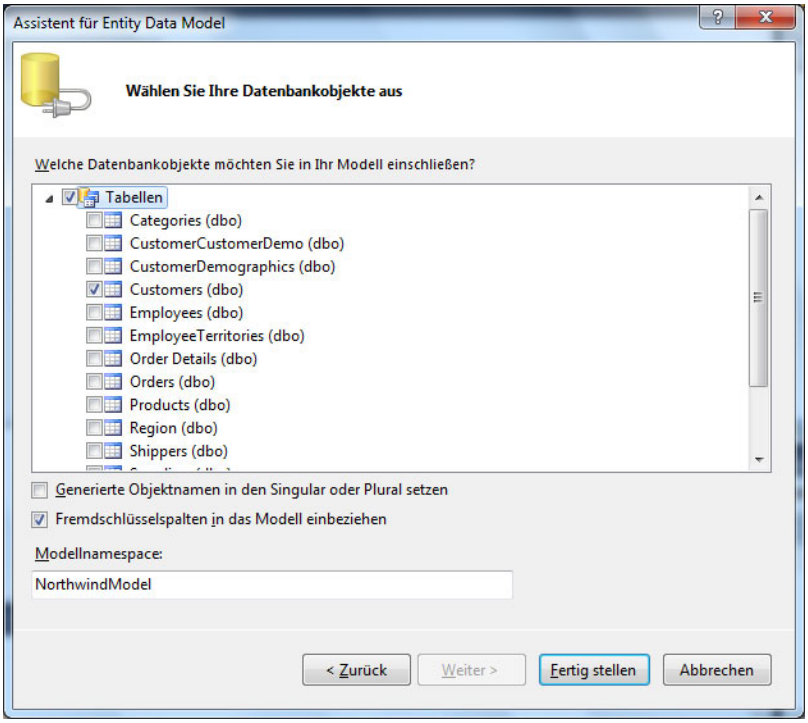


Abbildung 10.15
Auswahl der Datenbankobjekte



Danach wird Ihnen das Modell angezeigt und Sie können über das Menü *Ansicht* die Mapping-Details zu diesem Modell anzeigen, wie in Abbildung 10.16 illustriert.

Sollte sich das Datenmodell ändern, so können Sie sehr einfach über das Kontextmenü das *Modell aus der Datenbank aktualisieren* lassen.

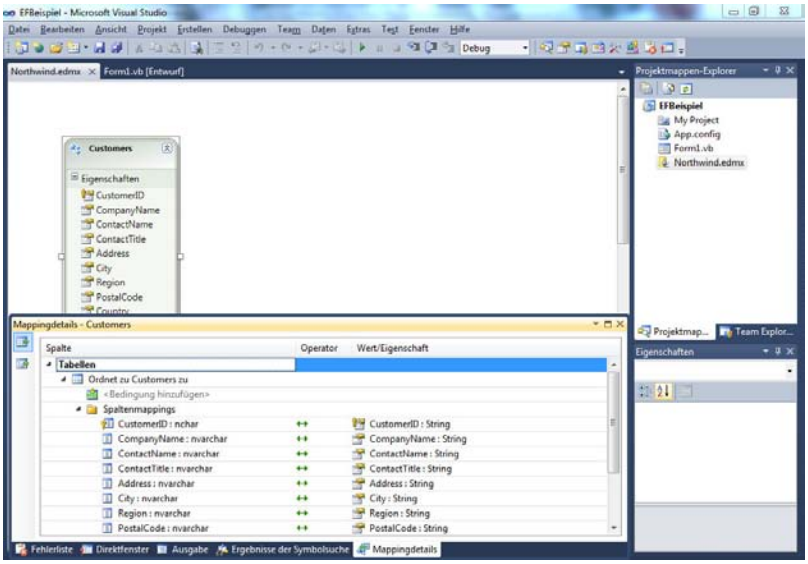


Abbildung 10.16
Mapping-Details

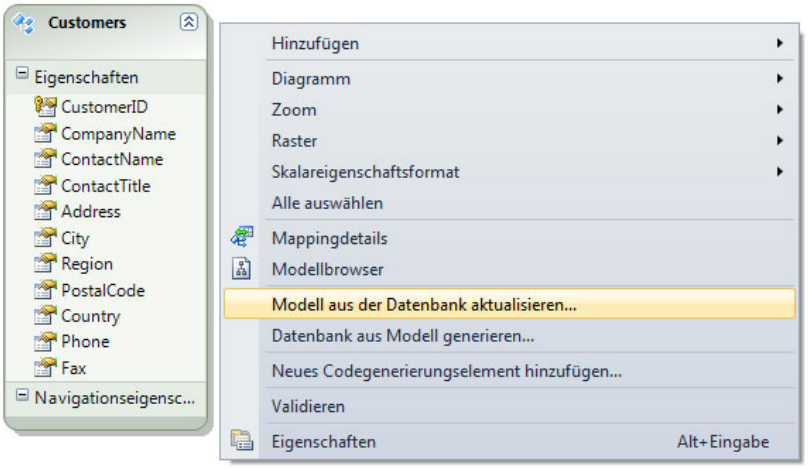


Abbildung 10.17
Aktualisierung des Modells aus der Datenbank

Den Zugriff auf dieses Entitätsdatenmodell mittels *LINQ to Entities* wollen wir im *Form1* des Beispielprojekts zeigen.

Fügen Sie also dem Formular eine `DataGridView` hinzu.

Da wir nur deutsche Kunden sehen wollen, verwenden wir dazu folgenden LINQ-Ausdruck:

```
From c In entities.Customers Where c.Country = "Germany"
select c
```

Wobei die Variable `entities` eine Instanz vom Typ `NorthwindModel.NorthwindEntities` darstellt. Listing 10.12 zeigt den Code der gesamten Website.

Listing 10.12 LINQ to Entities

```
Private Sub Form1_Load(ByVal sender As Object, _
  E As EventArgs) Handles MyBase.Load
      Dim entities As New NorthwindEntities()
      DataGridView1.DataSource = From c In entities.Customers _
          Where c.Country = "Germany" _
          Select c
End Sub
```

Das Entity Framework bietet aber nicht nur die Möglichkeit, Daten zu lesen, sondern auch Änderungen, die im Modell durchgeführt wurden, in die Datenbank zu persistieren.

Die Methode `SaveChanges()` der entsprechenden Entitätsklasse schreibt alle vorgenommenen Änderungen in die Datenbank zurück.

Der folgende Codeausschnitt (Listing 10.13) zeigt, wie Sie einen neuen Kunden anlegen und diesen danach in der Datenbank speichern:

Listing 10.13 Anlegen eines neuen Kunden

```
Dim cus As New NorthwindModel.Customers()
cus.CustomerID = "AdWes"
cus.CompanyName = "Addison Wesley"
cus.City = "München"
cus.Country = "Germany"
entities.AddObject("Customers", cus)
entities.SaveChanges()
```

11

Deployment

Mit Visual Studio 2010 haben Sie verschiedene Möglichkeiten, Ihre Applikationen zu verteilen. Dazu zählen **Web**, **ClickOnce**, **Windows Installer** und eingeschränkt auch noch **XCopy**-Deployment. In diesem Kapitel will ich Ihnen zeigen, wie Sie mit ClickOnce und Windows Installer Ihre Anwendungen verteilen können.

11.1 ClickOnce

Bis zur Version 2.0 des .NET Framework waren alle .NET-Anwendungen isoliert. Das bedeutet, dass die Anwendungen lokal auf dem Computer installiert wurden, ohne dass eine Registrierung der DLL oder der zugehörigen Komponenten erforderlich war (solange Sie nicht COM-Komponenten in Ihre Applikation eingebunden haben). Dieses löste (wie bereits in Kapitel 1.3.4 besprochen) das Problem der **DLL Hell**. Mit vorigen .NET-Versionen (**No Touch Deployment**) war es bereits möglich, Anwendungen aus dem Internet auszuführen. Die Anwendung wurde in einer sogenannten Sandbox, isoliert vom Rechner, ausgeführt. Dabei war es egal, auf welcher Art von Webserver die Anwendung gespeichert war.

Diese Technologie ermöglichte es zwar, Anwendungen direkt aus dem Internet auszuführen, ohne diese erst vorher auf der Maschine zu installieren, jedoch brachte sie wegen Code Access Security nicht immer die gewünschte Produktivität. Um beispielsweise auf systeminterne Funktionen zuzugreifen, musste erst eine Security Policy erstellt werden, damit der Zugriff von der Sandbox auf den Rechner gestattet wurde. Daneben wurde die Ausführungsgeschwindigkeit wesentlich beeinträchtigt, da die benötigten Komponenten erst zur Laufzeit aus dem Netz geladen werden mussten.

Mit der ClickOnce-Installation hat das Microsoft-Entwicklerteam schließlich das Web mit der lokalen Maschine vereint. Die Anwendung läuft lokal auf Ihrem System, kann über das Web geladen und vor allem automatisch upgedatet werden, ohne dass der Benutzer Administratorrechte besitzen muss. Es gibt zusätzlich die Möglichkeit, die Applikationen online zu installieren. ClickOnce wurde im Übrigen mit dem .NET Framework 2.0 eingeführt.

ClickOnce-Installation hilft Ihnen dabei, Installationsroutinen zu erstellen und die Anwendung bei Ihren Anwendern immer auf dem aktuellsten Stand zu halten.

Dabei wird Ihr Rechner keiner Gefahr ausgesetzt, denn ClickOnce wird keine Dateien ändern, die nicht zur Applikation gehören.

Bei der Betrachtung von Webapplikationen und Windows-Applikationen gab es bislang immer ein Pro und Contra. Zum einen hatten Webapplikationen den Vorteil der einfachen Verteilung, jedoch auf Kosten von eingeschränkten User Interfaces. Bei Windows-Applikationen stellte sich dies genau andersherum dar. Einem sehr hohen Verteilungsaufwand, der natürlich auch sehr hohe Kosten nach sich zieht, standen komfortable Benutzeroberflächen entgegen. ClickOnce führt beide sich bislang ausschließenden Vorteile zusammen, einfache Verteilung und keine Beschränkungen bei der Benutzeroberfläche.

Nun sollten wir uns aber anschauen, wie Sie mittels ClickOnce eine Applikation verteilen können.

11.1.1 Installation einer ClickOnce-Applikation

ClickOnce ist vollständig in die Entwicklungsumgebung Visual Studio 2010 integriert.

Ich öffne nun eine bereits bestehende Applikation, die ich jetzt verteilen möchte. Im Menü ERSTELLEN gibt es einen Untermenüpunkt PROJEKTNAME VERÖFFENTLICHEN. Nach der Auswahl dieses Menüpunkts startet der *Webpublishing-Assistent* (siehe Abbildung 11.1).

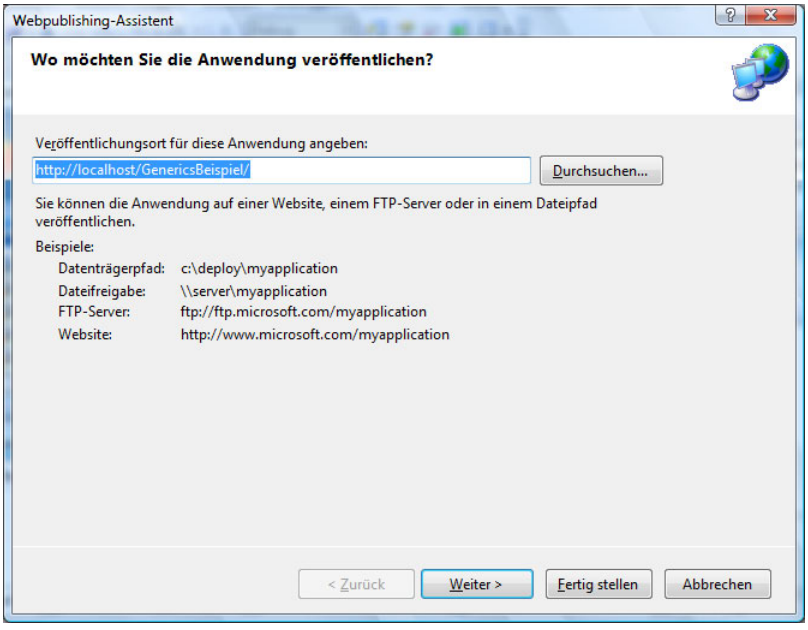


Abbildung 11.1
Starten des Webpublishing-Assistenten

Hier können Sie ein Verzeichnis auf einem Webserver angeben, von dem die Applikation später verteilt werden kann.

Im nächsten Schritt des Assistenten können Sie auswählen, ob die Applikation online und offline oder nur online verfügbar sein soll (siehe Abbildung 11.2). Anschließend wird noch eine Zusammenfassung angezeigt, welche Einstellungen Sie getroffen haben, bevor Sie mit FERTIG STELLEN die Veröffentlichung starten.

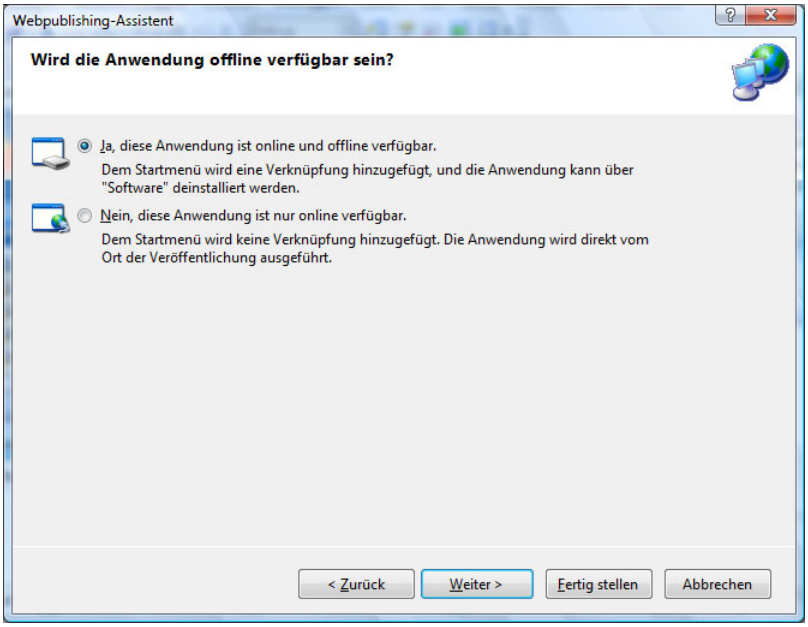


Abbildung 11.2
Einstellungen für die Verfügbarkeit der Applikation

Info

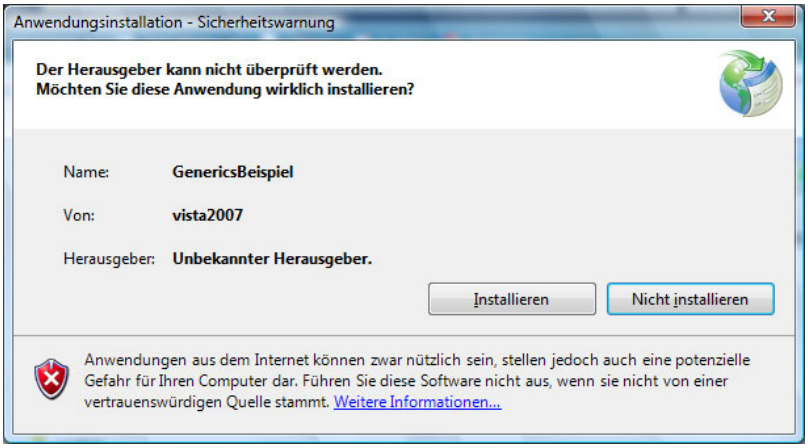
Mit ClickOnce installierte Applikationen können jederzeit über die SYSTEMSTEUERUNG wieder deinstalliert werden.

Nachdem die Veröffentlichung abgeschlossen ist, wurden alle für die Installation benötigten Dateien in das angegebene Verzeichnis auf dem Webserver erstellt oder kopiert.

Ebenso wird Ihr Standardbrowser gestartet und eine Datei *publish.htm* angezeigt.

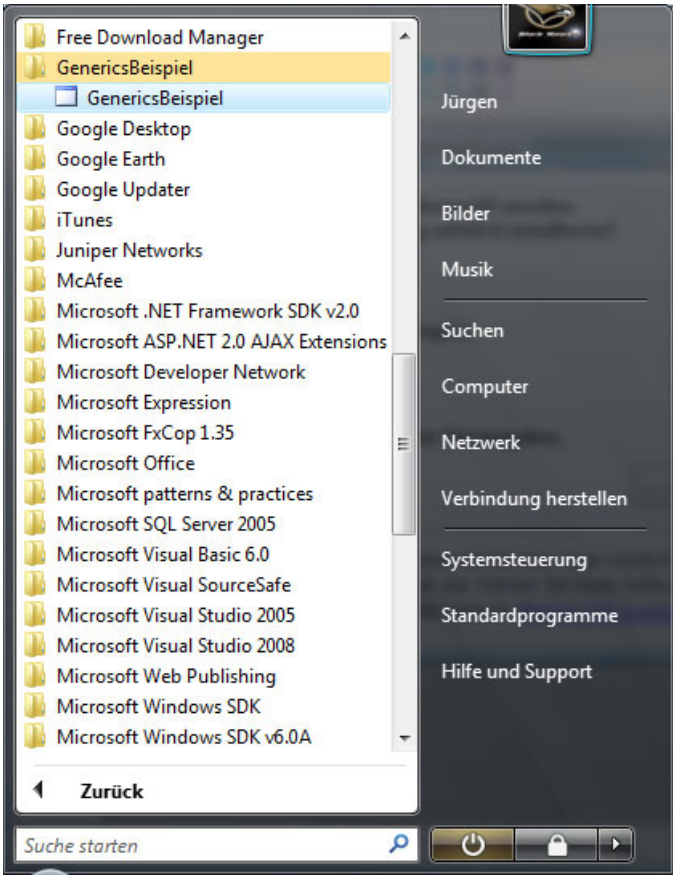
Nachdem Sie über den Button INSTALLIEREN die Installation gestartet haben, wird zuerst überprüft, ob die Voraussetzungen zur Installation auf dem Rechner erfüllt sind. Verläuft die Prüfung positiv, so erscheint der Dialog SICHERHEITSWARNUNG, den Sie in Abbildung 11.3 sehen. Dies geschieht jedoch nur, wenn die Anwendung auch offline verfügbar sein soll. Haben Sie bei der Option für die Verfügbarkeit den Standardwert auf nur online verfügbar geändert, so startet die Applikation ohne eine lokale Installation.

Abbildung 11.3
Sicherheitswarnung



Nach erfolgreicher lokaler Installation wird sofort im Anschluss daran auch die Anwendung gestartet. Im Startmenü wurde ein entsprechender Eintrag angelegt, wie Sie in Abbildung 11.4 sehen.

Abbildung 11.4
Startmenü



11.1.2 Update der Anwendung

Bislang wurde die Erstinstallation einer Anwendung beschrieben, doch jetzt gehen wir zum nächsten Schritt und wollen unsere Applikation aktualisieren.

Dies funktioniert prinzipiell genauso wie das Veröffentlichen der Anwendung. Sie müssen nur die Nummer der Veröffentlichungsversion erhöhen. Standardmäßig wird bei jedem neuen Veröffentlichen die Revisionsnummer automatisch inkrementiert.

Eine manuelle Anpassung der Veröffentlichungsversionsnummer können Sie unter PROJEKTEIGENSCHAFTEN auf dem Register VERÖFFENTLICHEN durchführen.

Sie müssen also für das Veröffentlichen des Updates lediglich im Menü ERSTELLEN wiederum den Eintrag PROJEKTNAME VERÖFFENTLICHEN aufrufen. Es wird die neue Version der Applikation veröffentlicht.

Achtung

Bitte verwechseln Sie nicht die Versionsnummer der Applikation mit der Veröffentlichungsversionsnummer. Diese beiden Nummern sind voneinander völlig unabhängig.

Nachdem die Veröffentlichung des Updates abgeschlossen wurde, finden Sie auf dem Webserver ein zweites Unterverzeichnis, in dem die Daten für das Update hinterlegt wurden. Der Verzeichnisname setzt sich dabei aus dem Projektnamen und der Veröffentlichungsversionsnummer zusammen.

Wenn jetzt auf dem Client die Applikation gestartet wird und eine Verbindung zum Webserver aufgebaut werden kann, dann erscheint der Update-Hinweis aus Abbildung 11.5. Mit der OK-Schaltfläche wird das Update durchgeführt.

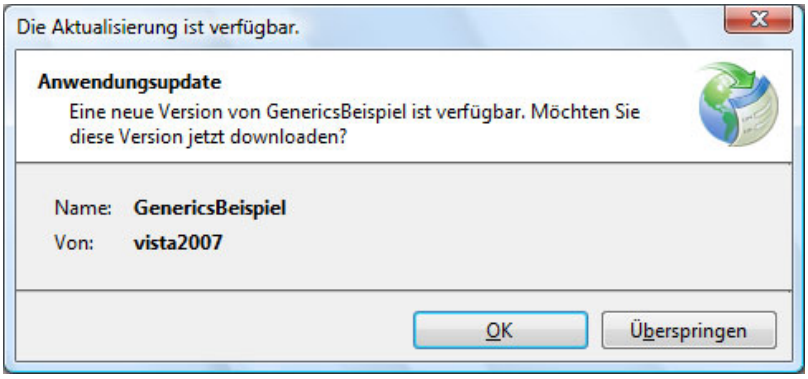


Abbildung 11.5
Update-Hinweis für ClickOnce-Applikationen

Bleibt noch die Frage, woher der Client weiß, dass ein neues Update zur Verfügung steht. Die Antwort: Im Verzeichnis auf dem Webserver gibt es unterschiedliche Veröffentlichungsmanifest-Dateien, die vom Client ausgewertet werden können.

Info

Veröffentlichungsmanifest-Dateien sind XML-Dateien.

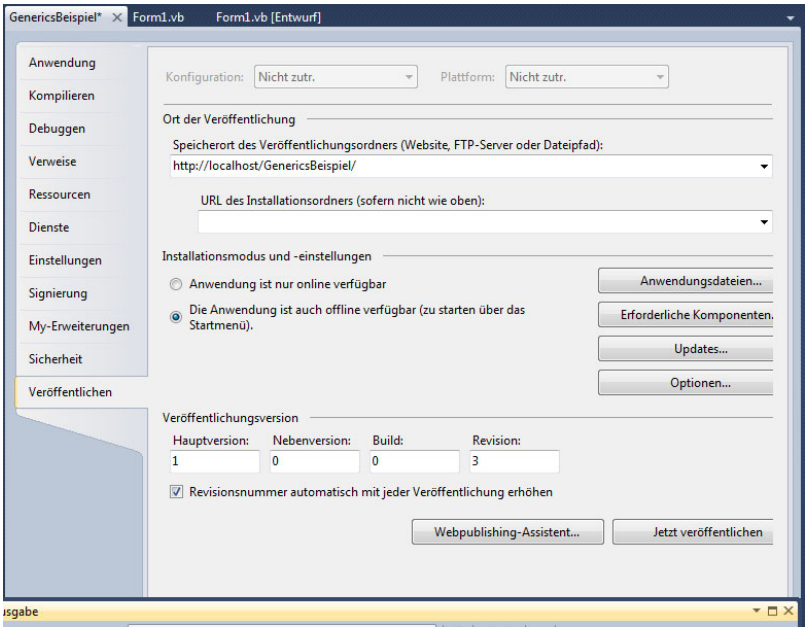
Dabei gibt es eine zentrale Deployment-Manifestdatei mit dem Namen *Projektname.application* und für jede veröffentlichte Version eine Manifestdatei, die sich aus dem Projektnamen und der Veröffentlichungsversion mit der Dateiendung *.application* zusammensetzt. Die zentrale Manifestdatei zeigt dabei auf die aktuelle Version.

11.1.3 Konfiguration von ClickOnce

Sowohl die Erstinstallation wie auch das Update haben wir mit den Standardeinstellungen durchgeführt. Jetzt will ich noch einen Blick auf die Konfigurationseinstellungen werfen, die wir für unsere ClickOnce-Installation vornehmen können.

Sämtliche Konfigurationseinstellungen zur Veröffentlichung finden wir im Register VERÖFFENTLICHEN in den Projekteigenschaften (siehe Abbildung 11.6).

Abbildung 11.6
Konfigurationsmöglichkeiten für ClickOnce



Die meisten Einstellungsmöglichkeiten sind meines Erachtens sehr intuitiv benannt, so dass ich auf eine ausführliche Beschreibung verzichte.

Der Veröffentlichungsort ist die Adresse, über die die Applikation veröffentlicht wird. Eine alternative URL für die Installation geben Sie dann an, wenn der Veröffentlichungsort ein Dateipfad oder ein FTP-Server ist.

Der Installationsmodus gibt an, ob die Applikation sowohl offline als auch online verfügbar ist oder ausschließlich nur online verfügbar sein soll.

Und am Ende der Seite sehen wir noch die Veröffentlichungsversionsnummer, bestehend aus Haupt-, Neben-, Erstellen- und Revisionsnummer. Das Kontrollkästchen für eine automatische Erhöhung der Revisionsnummer ist ebenso gesetzt.

Die beiden Buttons im unteren Teil des Dialogs starten zum einen den Assistenten, den ich zu Beginn des Kapitels beschrieben habe, und zum anderen wird eine neue Veröffentlichung gestartet, wie Sie sie auch aus dem Menü ERSTELLEN – PROJEKTNAME ERSTELLEN heraus aufrufen können.

Sehen wir uns aber noch die weiteren Dialoge an, die durch die Buttons auf der rechten Seite zu aktivieren sind.

Beginnen wollen wir mit dem Button ANWENDUNGSDATEIEN..., der uns zum Dialog *Anwendungsdateien* führt (siehe Abbildung 11.7).

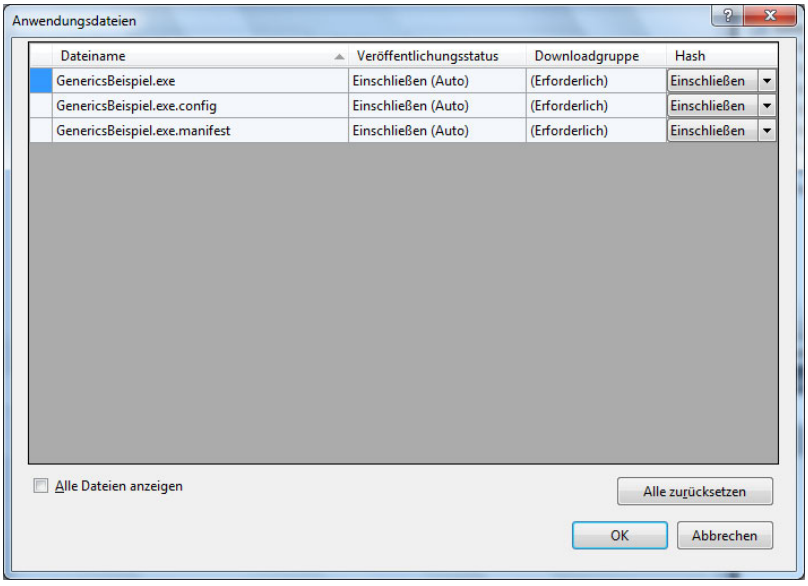


Abbildung 11.7
Anwendungsdateien einer ClickOnce-Installation

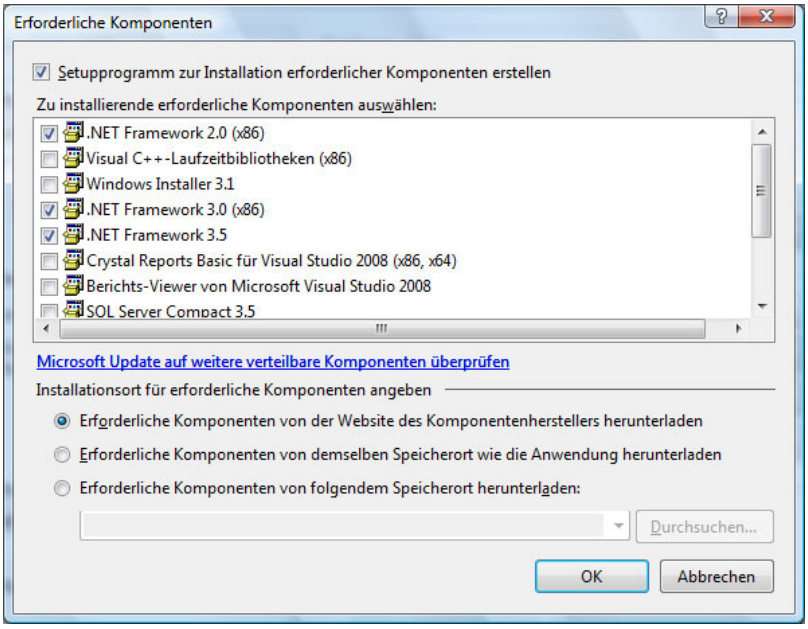
Unter bestimmten Umständen wollen Sie nicht alle Dateien aus dem *bin*-Verzeichnis des Projekts verteilen, da Sie manche nur zum Testen oder für Entwicklungszwecke benutzt haben. Diese Dateien können Sie an dieser Stelle vom Download ausschließen, indem Sie den Wert in der Spalte VERÖFFENTLICHUNGSSTATUS ändern.

Ansonsten finden Sie hier eine Übersicht über alle Dateien, die zum Client übertragen werden.

Der Abbildung können Sie entnehmen, dass die Debug (.pdb) und Dokumentationsdatei (.xml) nicht übertragen werden.

Der Button Erforderliche Komponenten... führt uns zum nächsten Dialog, den Sie in Abbildung 11.8 sehen.

Abbildung 11.8 Erforderliche Komponenten



In diesem Dialog können alle erforderlichen Komponenten angegeben werden, die benötigt werden, damit die Anwendung läuft. Das sind auch genau die Komponenten, deren Vorhandensein vor der eigentlichen Installation abgeprüft wird. Sie können dabei auch angeben, von welchem Speicherort die entsprechenden Komponenten heruntergeladen werden. Diese Komponenten werden in einem eigenen Installationsprogramm zur Verfügung gestellt, dem sogenannten **Bootstrapper**.

Über den Bootstrapper haben Sie somit auch die Möglichkeit, eigene Assemblies in den Global Assembly Cache zu installieren, was bei der normalen ClickOnce-Applikation nicht möglich ist.

Jedoch benötigen Sie für einen Bootstrapper auch Administratorrechte, da dieser über Windows Installer (*.msi*-Dateien) bereitgestellt wird.

Für die Dateien im Bootstrapper gibt es auf der *publish*-Webseite später einen eigenen Link, wie Sie in Abbildung 11.9 sehen.

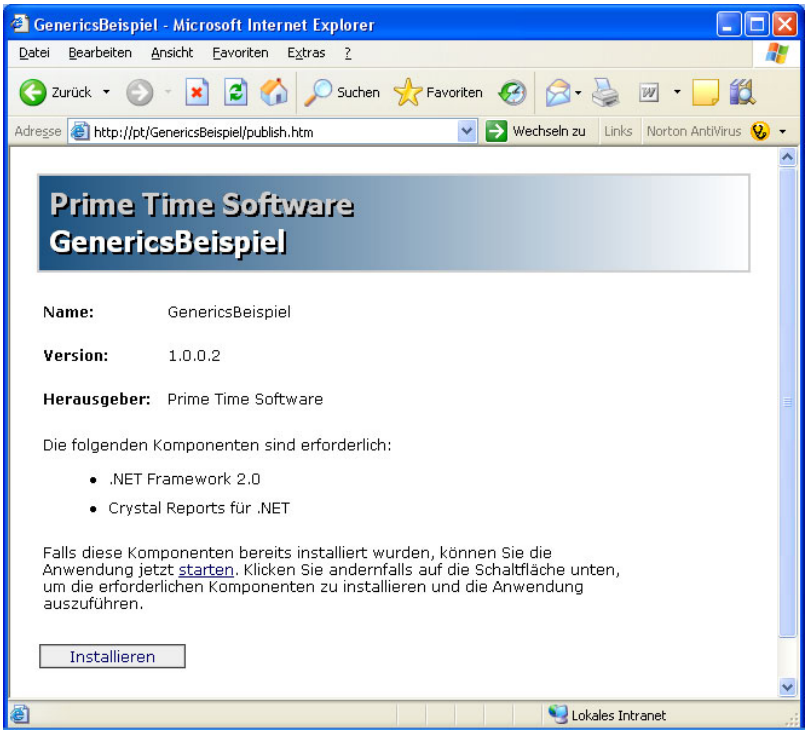


Abbildung 11.9
publish.htm mit erforderlichen Komponenten

Als Nächstes betrachten wir die Schaltfläche UPDATE... (siehe Abbildung 11.10).

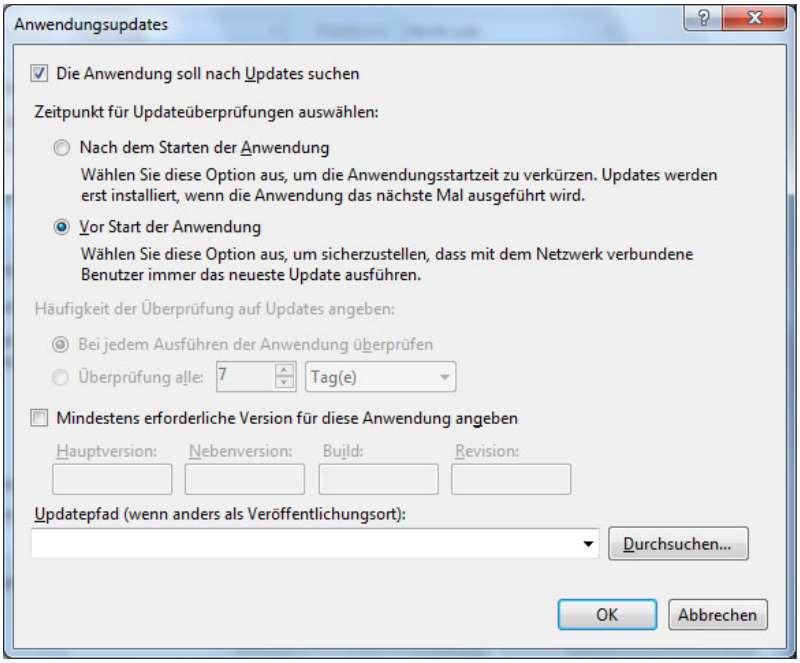


Abbildung 11.10
Anwendungsupdates

Die wichtigste Einstellung ist der Zeitpunkt, an dem die Anwendung nach Updates suchen soll. Die Standardeinstellung Vor Start der Anwendung gewährleistet, dass jeder Client mit der aktuellen Version arbeitet. Dies ist insbesondere dann wichtig, wenn z.B. Ihre Anwendung nicht mehr mit dem zugrunde liegenden Datenbankmodell auf dem Server kompatibel ist. Diese Einstellung hat jedoch den Nachteil, dass der Start der Applikation verzögert ausgeführt wird.

Die Alternative Nach dem Starten der Anwendung garantiert zwar einen performanteren Start der Applikation, Sie werden jedoch auf Updates erst beim nächsten Programmstart aufmerksam gemacht.

Außerdem können Sie noch einstellen, wie oft nach Anwendungen gesucht werden soll. Standardmäßig bei jedem Programmstart, Sie können aber auch Zeitintervalle auswählen. Wenn es nebensächlich ist, dass User nicht mit der aktuellen Version arbeiten, können Sie diese Option durchaus ändern, um langsame Programmstarts zu verhindern. Ich finde diese Einstellungsmöglichkeit eigentlich eher überflüssig, denn wenn es nicht wichtig ist, dass die Benutzer immer die aktuellste Version haben, dann kann ich die Überprüfung auf Aktualisierung auch im Hintergrund laufen lassen, nachdem mein Programm bereits gestartet wurde.

Ganz unten im Dialog können Sie noch einen Aktualisierungspfad angeben, wenn er nicht identisch mit dem Veröffentlichungsort ist.

Achtung

Seien Sie sich bewusst, dass die Einstellungen, die Sie hier vornehmen, für die nächste Version gelten und nicht für die aktuelle.

Außerdem sind diese Einstellungen nur sinnvoll, wenn die Option Die Applikation ist nur online verfügbar **nicht** gewählt ist, denn dann wird sowieso immer die aktuellste Version aus dem Web gestartet.

Über die Schaltfläche Optionen rufen Sie den Dialog Veröffentlichungsoptionen auf. Hier können Sie unter Sprache für Veröffentlichung die Sprache für die Benutzerschnittstelle, die während des Update-Vorgangs angezeigt wird, einstellen. Wenn Sie den Eintrag auf (Standard) lassen, werden dabei die Ländereinstellungen des Clients verwendet.

Unter Support-URL können Sie eine webbasierte Hilfeseite angeben, die auf der *publish*-Website als Link angezeigt wird.

Die Bereitstellungswebseite muss nicht zwingend *publish.htm* heißen, Sie können ihr in dem entsprechenden Textfeld auch einen anderen Namen zuweisen.

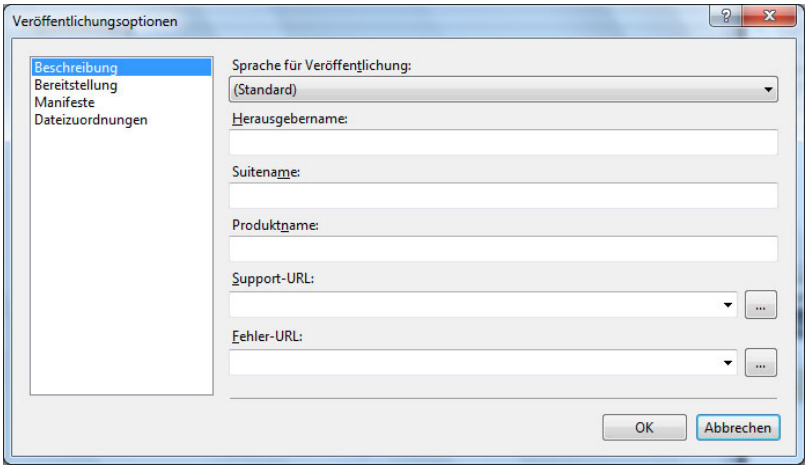


Abbildung 11.11
Veröffentlichungsoptionen

11.1.4 Sicherheitsüberlegungen für ClickOnce

Sie müssen sich jetzt noch darüber im Klaren sein, dass jegliche Art von Code, somit auch bösartiger Code, mittels ClickOnce verteilt und installiert werden kann.

Aus diesen Gründen läuft jede ClickOnce-Applikation in einer Sandbox, die in Abhängigkeit von Sicherheitseinstellungen auf den lokalen Rechnern unterschiedliche Rechte haben kann.

In Tabelle 11.1 sehen Sie eine Übersicht, innerhalb welcher Sicherheitszone Ihre Applikation in Abhängigkeit vom Veröffentlichungsort läuft.

Tabelle 11.1
Sicherheitszonen von ClickOnce-Applikationen

Veröffentlichungsort	Sicherheitszone
Von einer Internetseite gestartet	Internetzone
Von einer Internetseite installiert	Internetzone
Von einer Intranetseite installiert	Lokales Intranet
Von einem Netzlaufwerk installiert	Lokales Intranet
Von CD installiert	Lokaler Arbeitsplatz

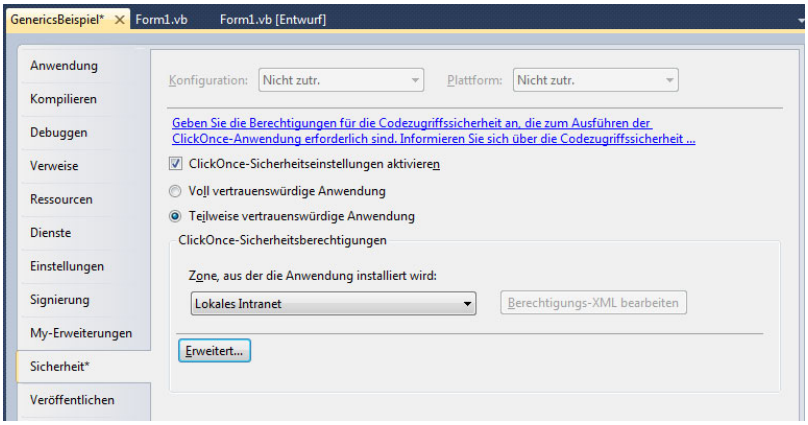
Bis auf die Zone *Lokaler Arbeitsplatz* besitzt keine weitere Sicherheitszone FullTrust-Level. Das bedeutet, die Anwendungen laufen mit einem eingeschränkten Rechteset und dürfen zum Beispiel keine Datenbankverbindungen öffnen oder auf das lokale Dateisystem zugreifen.

Standardmäßig beantragt die Applikation bei der Erstinstallation FullTrust-Level. Sie erinnern sich vielleicht noch an die Sicherheitswarnung. Dieser Dialog, auch wenn man es auf den ersten Blick nicht gleich sieht, weist Sie auf die Gefahren hin, und wenn Sie den Button Installieren anklicken, geben Sie dieser Anwendung auch Ihr volles Vertrauen.

Sie können aber auch für Ihre Anwendung ein Rechteset beantragen, dass der Applikation genau die Rechte einräumt, die diese zum korrekten Ablauf benötigt.

Dazu öffnen Sie unter Projekteigenschaften die Registerkarte Sicherheit (siehe Abbildung 11.12).

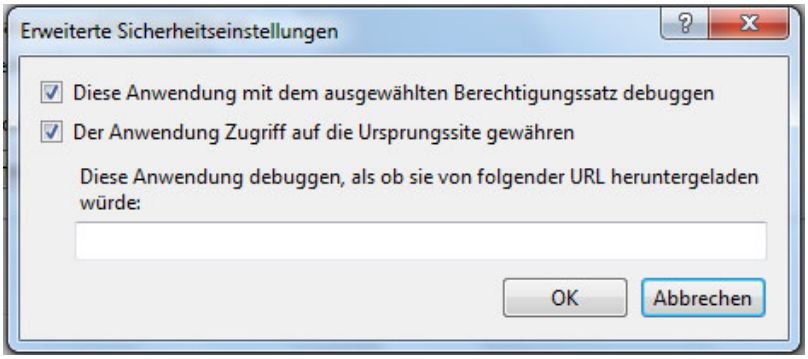
Abbildung 11.12
Sicherheitseinstellungen für ClickOnce



In Abbildung 11.12 habe ich den Standardwert bereits auf Teilweise vertrauenswürdige Anwendung umgestellt. Erst danach werden die weiteren Dialogelemente verfügbar. Sie können jetzt die Zone angeben, die dem Veröffentlichungsort Ihrer ClickOnce-Applikation entspricht (siehe Tabelle 11.1 »Sicherheitszonen von ClickOnce-Applikationen«).

Um zu überprüfen, ob die Rechte auch für die Anwendung ausreichend sind, klicken Sie auf die Schaltfläche Erweitert... Sie kommen dann zum Dialog Erweiterte Sicherheitseinstellungen (siehe Abbildung 11.13). In diesem Dialog können Sie die Applikation mit dem ausgewählten Berechtigungssatz starten. Sollten die Rechte ausreichen, so wird die Applikation fehlerfrei funktionieren. Haben Sie zu wenige Rechte ausgewählt, so wird eine Security-Exception als Laufzeitfehler auftreten. Dann sollten Sie noch weitere zusätzlich benötigte Berechtigungen beantragen.

Abbildung 11.13
Erweiterte Sicherheitseinstellungen für ClickOnce



Sobald die Applikation jedoch mehr Rechte braucht, als sie über die Sicherheitszone zugewiesen bekommt, muss sie diese vom Client durch Bestätigung des INSTALLIEREN-Buttons in dem Dialog SICHERHEITSWARNUNG (Abbildung 11.13) bekommen.

11.1.5 ClickOnce per Code beeinflussen

ClickOnce stellt zusätzlich noch eine Programmierschnittstelle zur Verfügung, um aus der Benutzeroberfläche heraus automatische Updates zu starten. Dies ist jedoch nur dann sinnvoll, wenn die Überprüfung auf Aktualisierung nicht bei jedem Programmstart durchgeführt wird (ich gehe davon aus, dass die wenigsten Applikationen 24 Stunden an sieben Tagen in der Woche laufen).

Um auf die ClickOnce-Klassen zugreifen zu können, müssen Sie in Ihrem Projekt einen Verweis auf die Bibliothek *System.Deployment.dll* setzen.

Außerdem sollten Sie eine Imports-Anweisung auf den Namespace `System.Deployment.Application` in Ihrem Code importieren.

Die wichtigste Klasse in diesem Namespace ist `ApplicationDeployment`.

Bevor Sie diese Klasse instanziieren, sollten Sie zuerst überprüfen, ob Ihre Applikation überhaupt eine ClickOnce-Applikation ist. Dies geschieht mittels der statischen Eigenschaft `IsNetworkDeployed`.

Über die Eigenschaft `CurrentDeployment` können Sie eine Instanz der Klasse `ApplicationDeployment` erstellen.

Mit der Methode `CheckForUpdate()` können Sie überprüfen, ob neue Updates vorhanden sind. Für diese Methode gibt es auch noch eine asynchrone Alternative: `CheckForUpdateAsync()`.

Zum tatsächlichen Download der neuen Sourcen können Sie dann letztendlich noch die Methode `Update()` der `ApplicationDeployment`-Klasse aufrufen beziehungsweise wiederum die asynchrone alternative `UpdateAsync()`.

Listing 11.1 zeigt ein Beispiel, wie Sie ein automatisches Update aus dem Code heraus aufrufen.

Listing 11.1
Update aus dem Programm heraus aufrufen

```
If ApplicationDeployment.IsNetworkDeployed Then
    Dim myUpdate As ApplicationDeployment = _
        ApplicationDeployment.CurrentDeployment
    If myUpdate.CheckForUpdate Then
        myUpdate.Update()
        MessageBox.Show("Update erfolgreich durchgeführt!")
    Else
        MessageBox.Show("Kein Update vorhanden!")
    End If
End If
```

11.2 Windows Installer

Auch wenn ClickOnce einige sehr schöne Verbesserungen gegenüber NoTouch-Deployment aufweist, sollte man den **Windows Installer** nicht abschreiben.

Denn es wird auch in Zukunft noch **Rich-Client-Applikationen** geben, bei deren Installation etwas tiefer ins System eingegriffen werden muss, als das ClickOnce überhaupt kann. Und für diese Zwecke eignet sich eben der Windows Installer besonders.

Außerdem bietet Windows Installer viel mehr Möglichkeiten, um Einfluss auf den Installationsprozess zu nehmen. So ist es möglich, die Benutzeroberfläche der Installationsprozesse anzupassen, Registrierungsschlüssel zu setzen und Aktionen nach der Installation aufzurufen.

Tipp

Windows Installer-Dateien können dem Bootstrapper von ClickOne hinzugefügt werden.

Windows Installer – die aktuelle Version ist 3.1 – ist ein Installationsdienst, der in den neuen Microsoft-Betriebssystemen (Windows XP, Windows Server 2003, Windows Server 2008, Windows Vista, Windows 7) integriert ist und auch im Visual Studio 2010 unterstützt wird.

Windows Installer benutzt *msi*-Dateien, um sämtliche Informationen sowie die Dateien für die Installation bereitzustellen. Eine wesentliche Verbesserung war bereits in der Version 3.0 die Möglichkeit, *msp*-Dateien zu erstellen, um damit Software-Patches einzuspielen, anstatt wieder ein gesamtes Installationspaket zu erstellen.

Wenn Sie in Visual Studio ein neues Projekt anlegen, finden Sie unter Installierte Vorlagen – Andere Projekttypen den Eintrag Setup und Bereitstellung – Visual Studio Installer. Wie Sie in Abbildung 11.14 sehen, gibt es für Setup-Projekte fünf unterschiedliche Projektvorlagen.

- Setup-Projekt

  Erstellt einen Installer für eine Windows-Applikation.

- Websetup-Projekt

  Erstellt einen Installer für eine Webapplikation.

- Mergemodulprojekt

  Mergemodule sind spezielle Installerdateien (**.msm*). Ein Mergemodul kann nicht alleine installiert werden, es muss immer in eine Windows Installer-Datei (**.msi*) integriert werden. Ein Mergemodul kann jedoch ganz spezielle Installationsanweisungen enthalten, die in verschiedenen Projekten gebraucht werden und somit nur einmal erstellt werden müssen. Zu einer Windows-Installerdatei können auch gleichzeitig mehrere Mergemodule hinzugefügt werden.

- Setup-Assistent

  Startet einen Assistenten, der den richtigen (hoffentlich!) Installertyp auswählt.

- CAB-Projekt
  Erstellt downloadbare CAB-Dateien.

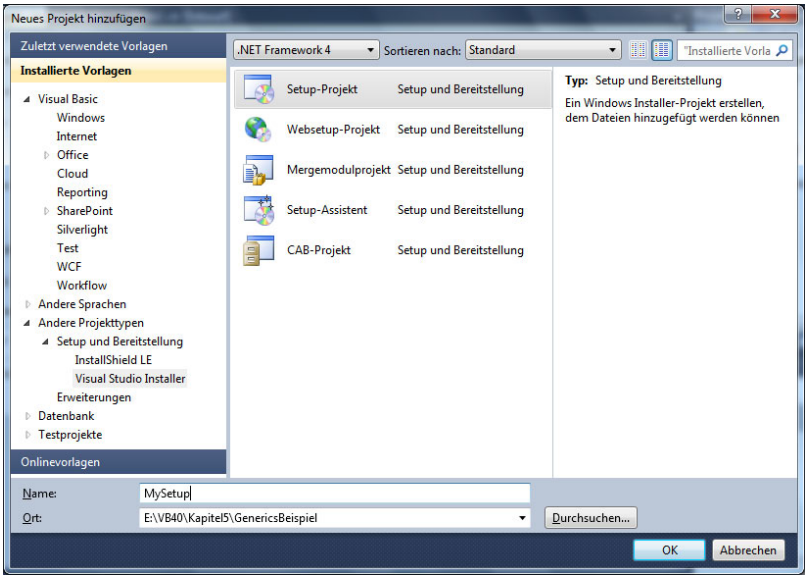


Abbildung 11.14
Installierte Vorlagen für Setup-Projekte

Wenn Sie nun ein Setup-Projekt ausgewählt haben, empfiehlt es sich, dass Sie die zu verteilende Applikation Ihrer Projektmappe hinzufügen. Sie können dann sehr einfach über das Kontextmenü HINZUFÜGEN – PROJEKTAUSGABE die benötigten Dateien für den Anwendungsordner auf dem Zielcomputer in Ihrem **Dateisystemeditor** zusammenstellen. Den entsprechenden Dialog sehen Sie in Abbildung 11.15.

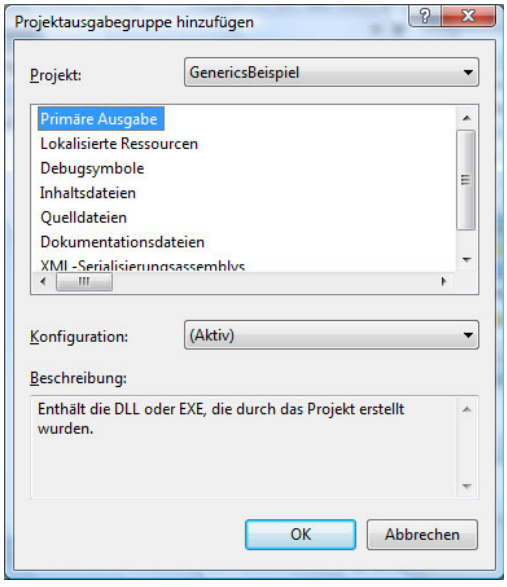
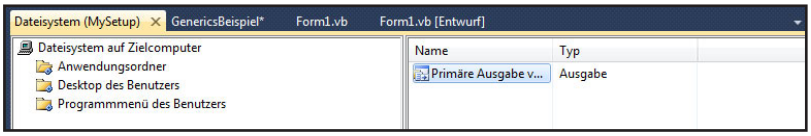


Abbildung 11.15
Projektausgabe hinzufügen

Damit ist eigentlich die Hauptaufgabe bereits erledigt, Sie können aber jetzt noch sehr viel Feinarbeit vornehmen, um die Installation zu konfigurieren.

Abbildung 11.16 zeigt Ihnen einen Ausschnitt aus dem Dateisystemeditor für die Konfiguration von Windows Installer-Paketen.

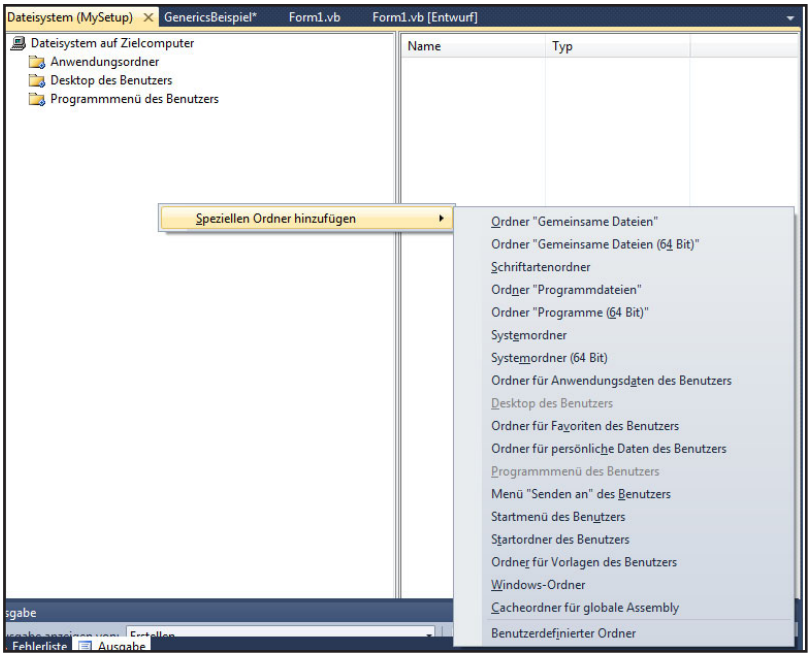
Abbildung 11.16
Dateisystem-Editor



Außer den Dateien im Anwendungsordner können Sie auch noch Shortcuts auf dem Desktop des Benutzers beziehungsweise in dessen Startmenü definieren.

Sie haben dabei auf der linken Seite die Möglichkeit, zusätzlichen speziellen Ordnern (GAC, Startmenü des Benutzers, Windows-Ordner, Schriftartenordner, Gemeinsame Dateien etc.) auf dem Zielcomputer Dateien hinzuzufügen (siehe Abbildung 11.17).

Abbildung 11.17
Hinzufügen von speziellen Ordnern



Außer dem Dateisystem-Editor haben Sie im Projektmappen-Explorer noch Symbole zum Aufruf anderer Editoren für den Installationsprozess:

- Registrierungs-Editor

  Hier können Sie Registrierungsschlüssel definieren, die während der Installation geändert oder hinzugefügt werden.

Abbildung 11.18
Registrierungseditor

- Dateityp-Editor

  Hier können Sie Dateitypen mit Ihrer Applikation verknüpfen und bestimmte Aktionen (Open, New, Print) definieren.

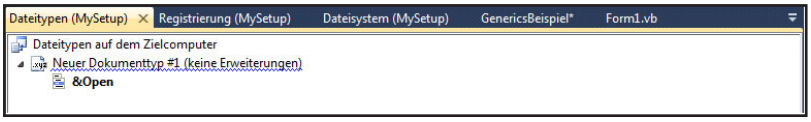


Abbildung 11.19
Dateitypeditor

- Benutzeroberflächen-Editor

  Hier können Sie die Benutzeroberfläche, die während des Installationsprozesses angezeigt wird, individuell anpassen.

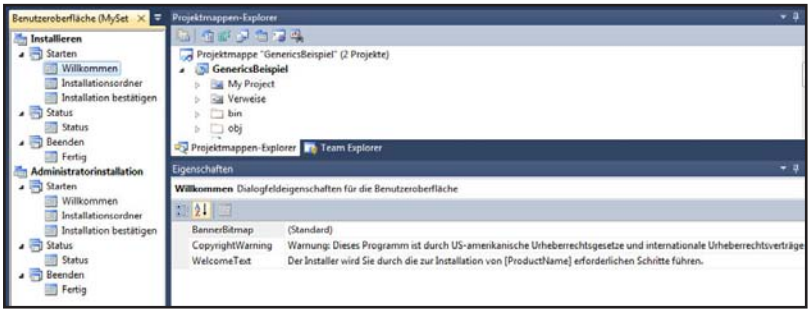


Abbildung 11.20
Benutzeroberflächeneditor

- Editor für benutzerdefinierte Applikationen

  Hier können Sie bestimmte Aktionen definieren, die nach dem Installationsprozess ausgeführt werden.

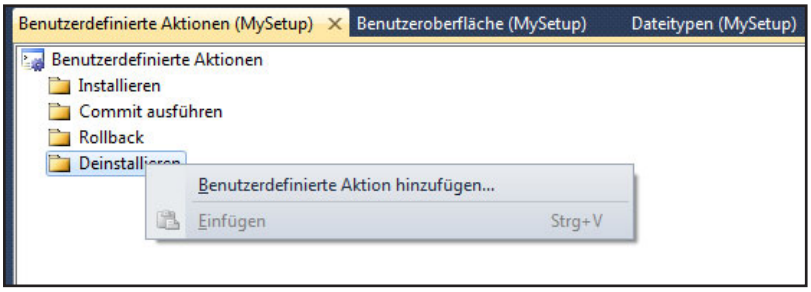
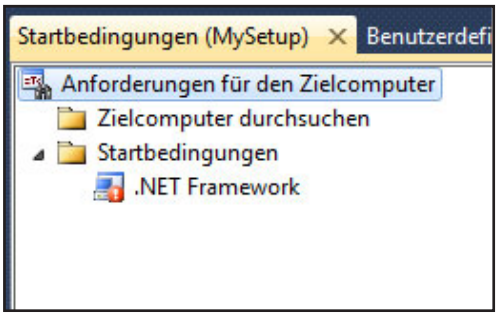


Abbildung 11.21
Editor für benutzerdefinierte Aktionen

- Editor für Startbedingungen

  Hier können Sie bestimmte Voraussetzungen (.NET Framework 4.0 bereits installiert) definieren, damit Ihr Setup-Projekt überhaupt auf dem Zielrechner gestartet wird.

Abbildung 11.22
Startbedingungseditor



Um das Setup zu erstellen, rufen Sie einfach im Menü ERSTELLEN den Eintrag <PROJEKTNAME> ERSTELLEN auf und Sie erhalten Ihre gewünschte Setup-Anwendung inklusive Windows Installer-Datei (*msi*).

12

Wichtige Basisklassen

.NET-Sprachen wie Visual Basic besitzen praktisch kaum eigene Funktionen, weil alle Funktionalität, die für die Programmierung benötigt wird, in der Base Class Library (BLC) des .NET Framework enthalten ist.

In der aktuellen Version 4.0 stehen über 45.000 Klassen zur Verfügung. 45.000 Klassen hört sich zunächst nach unendlich viel Arbeit an. Wohl niemand wird Zeit haben, sich mit wirklich allen Klassen zu beschäftigen. Viele davon sind jedoch so speziell, dass sie kaum ein Programmierer direkt ansprechen wird, und viele Klassen sind aufgrund Vererbung und Interfaces sehr ähnlich aufgebaut.

Tabelle 12.1 gibt einen kleinen Überblick über die wichtigsten Namespaces, in denen die entsprechenden Basisklassen zu finden sind.

Tabelle 12.1
Wichtige Namespaces für Basisklassen

Namespace	Verwendung
System.IO	Stellt Klassen für den Zugriff auf das Dateisystem bereit.
System.IO.Compression	Enthält Klassen zur Komprimierung/Dekomprimierung von Texten.
System.IO.Ports	Stellt Klassen für den Zugriff auf die serielle Schnittstelle und daran angeschlossene Geräte bereit.
System.Configuration	Stellt Klassen für den Zugriff auf die Konfigurationsdateien im XML-Format der eigenen Anwendung bereit.
System.Security.Cryptography	Enthält Klassen zum Ver- und Entschlüsseln von Daten.
System.Resources	Stellt Klassen für den Zugriff und die Erstellung von Ressourcen-Dateien, die zum Beispiel für die Lokalisierung genutzt werden können, bereit.
System.Net.Mail	Stellt Klassen für das Senden von E-Mails aus der eigenen Anwendung heraus bereit.

Tabelle 12.1 (Forts.) Wichtige Namespaces für Basisklassen

Namespace	Verwendung
System.Diagnostic	Stellt Klassen zur Verfügung, die Anwendungsdaten ermitteln und auswerten.
System.Drawing	Stellt Zeichenobjekte für GDI+ zur Verfügung.
System.Threading	Stellt Klassen für Multithreading und Synchronisation zur Verfügung.
System.Runtime.Serialization	Bietet Klassen für Serialisierung und Formatierung.

12.1 Dateizugriff

Die Funktion des Dateizugriffs auf das Dateisystem ist bereits seit den ersten Versionen von Visual Basic vorhanden. Mit Einführung des .NET Framework wurde das Konzept des Dateizugriffs jedoch komplett neu geordnet.

Das .NET Framework 4.0 bietet verschiedene Klassen, um auf unterschiedliche Weise auf Ordner und Dateien zuzugreifen. Im Folgenden lernen Sie den Umgang mit folgenden Klassen kennen:

- FileStream
- MemoryStream
- Directory
- File

12.1.1 FileStream

Die `FileStream`-Klasse erlaubt es Ihnen, Daten zu lesen, zu schreiben, zu öffnen oder zu schließen. Hierbei können die Daten synchron oder asynchron geschrieben oder gelesen werden.

In der Tabelle 12.2 finden Sie die meist benutzten Eigenschaften und Methoden inklusive des dazugehörigen Zwecks.

Tabelle 12.2 Eigenschaften und Methoden der FileStream-Klasse

Eigenschaft/ Methode	Zweck
`CanRead`	Gibt an, ob die Datei Lesezugriff besitzt.
`CanWrite`	Gibt an, ob die Datei Schreibzugriff besitzt.
`Name`	Der Name der Datei
`ReadTimeout`	Gibt die Zeit in Millisekunden zurück, welche die Anwendung wartet, bis sie einen Timeout-Fehler fürs Lesen der Datei ausgibt.
`WriteTimeout`	Gibt die Zeit in Millisekunden zurück, welche die Anwendung wartet, bis sie einen Timeout-Fehler fürs Schreiben der Datei ausgibt.
`BeginRead()`	Beginnt, die Datei asynchron zu lesen.

Eigenschaft/ Methode	Zweck
BeginWrite()	Beginnt, die Datei asynchron zu schreiben.
Read()	Liest Bytes aus dem Stream aus.
ReadByte()	Liest ein Byte aus der aktuellen Position des Stream.
Write()	Schreibt ein Byte in einen Stream.
WriteByte()	Schreibt ein Byte an die aktuelle Position des Stream.
Close()	Schließt den Dateizugriff und gibt diese frei.
Seek()	Durchsucht, falls möglich, die Datei.

Tabelle 12.2 (Forts.) Eigenschaften und Methoden der FileStream-Klasse

12.1.2 MemoryStream

Im Gegensatz zum FileStream, wo die Daten auf einem lokalen oder Netzwerkdatenträger gespeichert werden, schreibt der MemoryStream die Daten in den lokalen Arbeitsspeicher. Dies schafft besonders bei temporären Dateien einen erheblichen Geschwindigkeitsvorteil. Die Daten werden direkt beim Erstellen des MemoryStream in ein Array im Arbeitsspeicher abgelegt und sind direkt abrufbar. Listing 12.1 verdeutlicht, wie Daten in einem MemoryStream geschrieben und gelesen werden:

Listing 12.1 MemoryStream schreiben und lesen

```
Public Sub memorystream()
    Dim mstream As New IO.MemoryStream()
    Dim stw As New IO.StreamWriter(mstream)
    Dim str As New IO.StreamReader(mstream)
    'Stream in Arbeitsspeicher schreiben
    stw.Write("Ich bin nur im Speicher")
    stw.Flush()

    'Position lesen und Daten wieder auslesen
    mstream.Seek(0, IO.SeekOrigin.Begin)
    System.Console.WriteLine(str.ReadToEnd())
    str.Close()
End Sub
```

12.1.3 Weitere Klassen in System.IO

Mit den StreamReader- und StreamWriter-Klassen können Sie zwar Dateien schreiben und lesen, bekommen aber keine Informationen über die Dateien selbst. So ist es nur sehr umständlich möglich, beispielsweise die Dateigröße durch die StreamReader-Klasse zu berechnen (nämlich die Datei einzulesen, Zeichen zu zählen, umzuwandeln, auszugeben). Der System.IO-Namespace ermöglicht es, mit relativ einfach zu verstehenden und anwendbaren Befehlen Informationen über Dateien und Ordner zu erlangen. Fast alle Informationen, die Sie als Entwickler vom Dateisystem benötigen, lassen sich mit den statischen Methoden der Klassen Directory und File abfragen.

In Tabelle 12.3 und Tabelle 12.4 sehen Sie alle wichtigen Methoden der `Directory`- und der `File`-Klasse:

Tabelle 12.3 Methoden der Directory-Klasse

Methode	Zweck
`CreateDirectory()`	Erstellt einen Ordner.
`Delete()`	Löscht einen Ordner und die enthaltenen Dateien.
`Exists()`	Prüft, ob ein Ordner existiert, und liefert einen `Boolean`-Wert zurück.
`GetCreationTime()`	Liefert das Erstelldatum des Ordners.
`GetCurrentDirectory()`	Gibt das aktuelle Verzeichnis wieder.
`GetDirectories()`	Liefert alle Namen der Unterordner in einem Array.
`GetFiles()`	Liefert alle Dateien eines Ordners in einem Array.
`GetLastAccessTime()`	Gibt das Datum des letzten Zugriffs auf einen Ordner zurück.
`Move()`	Verschiebt ein Verzeichnis einschließlich des Inhalts.

Tabelle 12.4 Methoden der File-Klasse

Methode	Zweck
`AppendAll()`	Falls die Datei bereits vorhanden ist, wird Text angefügt. Ansonsten wird die Datei erstellt und Text eingefügt.
`Copy()`	Kopiert eine Datei an einen anzugebenden Ort.
`Create()`	Erstellt eine Datei.
`Delete()`	Löscht die Datei.
`Exists()`	Prüft, ob die Datei existiert, und liefert einen `Boolean`-Wert zurück.
`GetCreationTime()`	Liefert das Erstelldatum der Datei.
`GetLastWriteTime()`	Liefert das Datum des letzten Zugriffs.
`Move()`	Verschiebt die Datei.
`Open()`	Öffnet die Datei. Mit Parametern kann angegeben werden, in welchem Modus die Datei geöffnet werden soll.
`ReadAllText()`	Liest den gesamten Text einer Datei ein und gibt ihn als String zurück.
`WriteAllText()`	Schreibt oder überschreibt einen String in einer Datei.

Weiter gibt es folgende wichtige Klassen wie `DirectoryInfo`, die einen Ordner mit all seinen Eigenschaften und Möglichkeiten widerspiegelt, `FileInfo`, das Gegenstück für Dateien von `DirectoryInfo`, und die `FileSystemWatcher`-Klasse, die Ordner oder Dateien auf Änderungen überwacht. Beispiele zum Gebrauch der einzelnen Klassen finden Sie im nächsten Abschnitt.

`DirectoryInfo` und `FileInfo` beinhalten sehr ähnliche Methoden wie `Directory` und `File`, jedoch als Instanzmethoden.

Spezielle Ordner

Visual Basic 10 erlaubt es, in Verbindung mit dem `My-Namespace` einfach auf von Windows festgelegte spezielle Ordner zuzugreifen. Sie erhalten Zugriff auf folgende Ordner über `My.Computer.FileSystem.SpecialDirectories`:

- AllUserApplicationData
- CurrentUserApplicationData
- Desktop
- MyDocuments
- MyMusic
- MyPictures
- ProgramFiles
- Programs
- Temp

12.2 Anwendungen mit System.IO

Um die Vielfalt der Zugriffsmöglichkeiten auf Daten im .NET Framework 4.0 zu zeigen, stellen wir hier einige Beispiele vor, die Lese- und Schreibzugriff über verschiedene Funktionen ermöglichen.

Neben der Sammlung von Informationen sind das Kopieren, Verschieben oder Löschen von Dateien und Ordnern die meist verwendeten Operationen, die vom Anwender durchgeführt werden. Die Namen der entsprechenden Funktionen sind durchweg selbst erklärend, was die Handhabung sehr einfach macht:

- CopyFile()
- MoveFile()
- RenameFile()
- DeleteFile()

Die Methoden für Ordner heißen:

- CopyDirectory()
- MoveDirectory()
- RenameDirectory()
- DeleteDirectory()

Der Einsatz dieser Methoden kann mit dem Code aus Listing 12.2 verdeutlicht werden. Hierbei wird zunächst mit einer einfachen `If`-Abfrage geprüft, ob der Zielordner bereits vorhanden ist. Falls das nicht der Fall ist, wird der Ordner erstellt und danach der temporäre Ordner kopiert. Es ist zwingend erforderlich, den `Overwrite`-Parameter anzugeben. Dieser `Boolean`-Wert legt

fest, ob Dateien oder Ordner, die bereits im Zielverzeichnis vorhanden sind, überschrieben werden sollen. Falls der `Boolean`-Wert auf `False` gesetzt wurde und die Datei oder der Ordner bereits vorhanden ist, wird es zu einer Fehlermeldung kommen.

Listing 12.2 Dateien und Ordner kopieren

```
Public Sub OrdnerKopieren()
   If Not Directory.Exists("C:\Backup") Then
     Directory.CreateDirectory("C:\Backup")
   End If
   My.Computer.FileSystem.CopyDirectory _
     (My.Computer.FileSystem.SpecialDirectories.Temp, _
      "C:\Backup", True)
End Sub
```

Falls Sie Ihren Anwendern die Möglichkeit bereitstellen möchten, bereits vorhandene Dateien mit einem Auswahldialog (wie man ihn aus dem Windows Explorer kennt) überschreiben zu lassen, können Sie dieses ebenfalls realisieren. Dazu stehen Ihnen in der Toolbox im Register DialogFelder mehrere Steuerelemente zur Verfügung.

Das folgende Beispiel in Listing 12.3 erstellt zunächst eine Datei und verschiebt diese Datei dann in einen anderen Ordner. Die ursprüngliche Datei wird dann gelöscht.

Listing 12.3 Mehrere File-Operationen

```
Public Sub VerschiedeneOperationen()
  Dim text As String = "Lesen macht Spaß"
  Dim quellDatei As String = "C:\Texte\texte.txt"

  Dim zielVerzeichnis As String = "C:\Ziel\"
  File.WriteAllText(quellDatei, text)

  'Ziel Verzeichnis erstellen
  If Not Directory.Exists(zielVerzeichnis) Then
    Directory.CreateDirectory(zielVerzeichnis)
  End If
  'Datei verschieben
  Try
    File.Move(quellDatei, zielVerzeichnis)
  Catch ex As Exception

  End Try
  File.Delete(quellDatei)
  Console.WriteLine("Aufgaben erledigt")
End Sub
```

Die Klasse `DriveInfo`-Klasse steht Ihnen zur Verfügung, um Informationen über Laufwerke auszulesen.

Folgende Anwendung liest alle Informationen über Laufwerke aus und speichert den Inhalt in einer Textdatei im *Eigene Dateien*-Ordner (unter Windows Vista und Windows 7 heißt der entsprechende Ordner *Dokumente*). Bei dem Codebeispiel wird unterschieden, ob ein Medium in den Laufwerken vorhanden ist oder nicht. Falls im DVD/CD-Laufwerk kein Medium vorhanden ist, können natürlich keine Informationen über die Größe ermittelt werden. Hierzu bietet die `DriveInfo`-Klasse die Eigenschaft `IsReady`.

Listing 12.4
Laufwerksinformationen ermitteln und speichern

```
Public Sub LaufwInfo()
  Dim text As String = "Starte Anwendung" & vbNewLine
  For Each drive As DriveInfo In DriveInfo.GetDrives()

    text &= drive.Name & vbNewLine

    If drive.IsReady Then

      text &= vbTab & "Bezeichnung: " & drive.VolumeLabel & _
           vbNewLine
      text += vbTab & "Gesamt Größe: " & drive.TotalSize & _
           vbNewLine
      text += vbTab & "Gesamt Größe in MB: " & _
           (drive.TotalSize / 1024).ToString  & vbNewLine
      text += vbTab & "Freier Speicherplatz:" & _
        drive.AvailableFreeSpace & vbNewLine
      text += vbTab & "Speicherplatz in MB: " & _
       (drive.TotalFreeSpace / 1024).ToString & vbNewLine
      text += vbTab & "Formatierung: " & _
        drive.DriveFormat.ToString() & vbNewLine
     End If
   Next
   File.WriteAllText _
    (My.Computer.FileSystem.SpecialDirectories.MyDocuments & _
     "\test.txt", text)
End Sub
```

Das Ergebnis könnte dann wie folgt in der *test.txt*-Textdatei aussehen:

Starte Anwendung

C:

Bezeichnung: OS

Gesamt Größe: 146486059008

Gesamt Größe in MB: 143052792

Freier Speicherplatz: 14100467712

Speicherplatz in MB: 13769988

Formatierung: NTFS

D:

Bezeichnung: RECOVERY

Gesamt Größe: 10737414144

Gesamt Größe in MB: 10485756

Freier Speicherplatz: 7063302144

Speicherplatz in MB: 6897756

Formatierung: NTFS

E:

F:

Mit dem `FileInfo`-Objekt können Sie nun Informationen über einen Ordner oder eine Datei erhalten. Sie legen hierbei eine zu prüfende Datei fest, in unserem Beispiel in Listing 12.5 wird es die *test.txt*-Datei sein, die gerade mit den Informationen über das Laufwerk gefüllt wurde. Die Informationen werden wiederum in einer neuen Datei gespeichert.

Listing 12.5 Informationen über eine Datei abrufen

```
Public Sub pruefen()
   Dim datei As FileInfo
   Dim text As String = ""
   Dim fileName As String = _
     My.Computer.FileSystem.SpecialDirectories.MyDocuments & _
        "\test.txt"
   If File.Exists(fileName) Then
     datei = New FileInfo(fileName)
     text &= "Erstellt: " & datei.CreationTime & vbNewLine
     text &= "Letzter Zugriff: " & datei.LastAccessTime & _
          vbNewLine
     text += "Als letztes geändert: " & datei.LastWriteTime & _
          vbNewLine
     text += "Größe der Datei in Bytes: " & _
          datei.Length.ToString & vbNewLine
   Else
      text = "Keine Informationen vorhanden"
   End If
   File.WriteAllText _
     (My.Computer.FileSystem.SpecialDirectories.MyDocuments & _
       "\informationen.txt", text)
End Sub
```

Die Ausgabe könnte dann wie folgt aussehen:

Erstellt: 17.03.2008 19:07:48

Letzter Zugriff: 17.03.2008 19:07:48

Als letztes geändert: 17.03.2008 19:25:16

Größe der Datei in Bytes: 384

Die Sammlung von Informationen ist übrigens mit jeder Datei möglich. Hierbei ist es egal, ob Sie Informationen über eine *.exe-, .doc-, .xls-* oder eben eine Textdatei abrufen möchten.

12.2.1 In Ordnern nach Dateien suchen

Es ist auch möglich, nach Ordnern und Dateien auf Laufwerken zu suchen. Hierzu stellt die Klasse `Directory` die Funktion `GetFiles()` zur Verfügung. Es kann festgelegt werden, ob nur der Hauptordner oder auch Unterordner durchsucht werden sollen. Die Ergebnisse können beim Beenden der Suche als String ausgegeben werden. Leider stehen die Ergebnisse nur nach Beendigung der Suche zur Verfügung.

Im folgenden Beispiel werden mit der Wildcard *.* alle Dateien im Temp-Verzeichnis durchsucht und als Text in der Datei *suche.txt* gespeichert.

```
Public Sub Suche()
  Dim datei As FileInfo
  Dim text As String = ""

  For Each fileName As String In Directory.GetFiles _
    ("C:\temp", "*.*", SearchOption.TopDirectoryOnly)

    datei = New FileInfo(fileName)
    text &= datei.Name & " im Ordner " & _
      datei.Directory.Name & vbNewLine
    text &= "Erstellt: " & datei.CreationTime & vbNewLine
    text &= "Letzter Zugriff: " & _
      datei.LastAccessTime & vbNewLine
    text &= "Größe der Datei in Bytes: " & _
      datei.Length & vbNewLine & vbNewLine
  Next
    File.WriteAllText _
     (My.Computer.FileSystem.SpecialDirectories.MyDocuments & _
       "\suche.txt", text)

End Sub
```

Listing 12.6
Durchsuchen von Ordnern und Unterordnern

Einen Teil des Ergebnisses sehen Sie in Abbildung 12.1.

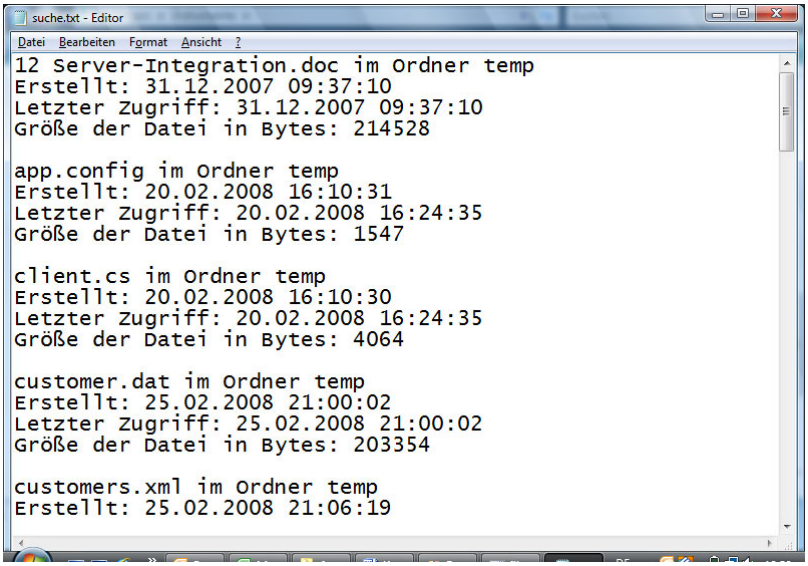


Abbildung 12.1
Inhalt der Datei suche.txt

12.2.2 Daten komprimieren und dekomprimieren

Durch immer größere werdende Ressourcen an Speicherplatz ist es heute kaum noch eine Frage, wie groß Dateien sind. Und trotzdem kann es sich lohnen, Daten zu komprimieren. Gehen wir einmal davon aus, dass Sie eine sehr große Datei auf einen anderen Server kopieren müssen. Dies würde ohne Komprimierung nicht nur lange dauern, sondern auch unnötige Bandbreite kosten.

Das .NET Framework 4.0 bietet hierfür mit dem Namespace `System.IO.Compression` Abhilfe. Hier können Sie unter anderem auf die Klasse `GZipStream` zum Komprimieren und Dekomprimieren zurückgreifen. Natürlich gehen bei dieser Art der Komprimierung keinerlei Daten verloren.

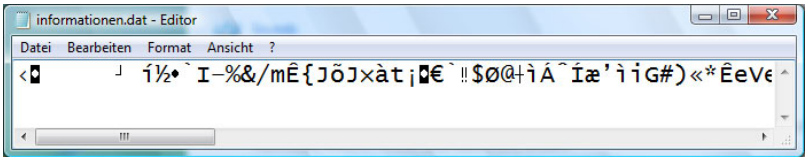
Bei der Anwendung sollten Sie darauf achten, dass Sie nicht direkt komprimierte Dateien schreiben können. Sie müssen zunächst mit der normalen `Stream`-Klasse Daten verfassen, um diese dann, wie in Listing 12.7 dargestellt, zu komprimieren.

Listing 12.7 Komprimieren einer Textdatei

```
Public Sub compression()
  'Quelldatei lesen
  Dim quellDatei As String = My.Computer.FileSystem. _
    SpecialDirectories.MyDocuments & "\informationen.txt"
  Dim stream As New FileStream(quellDatei, FileMode.Open)
  Dim fileBytes(Convert.ToInt32(stream.Length - 1) As Byte
  stream.Read(fileBytes, 0, fileBytes.Length)
  stream.Close()

   'Neue komprimierte Datei erstellen
  Dim zielDatei As String = My.Computer.FileSystem. _
    SpecialDirectories.MyDocuments & "\informationen.dat"
  Dim writer As New FileStream(zieldatei, FileMode.Create)
  Dim compStream As New Compression.GZipStream _
   (writer, IO.Compression.CompressionMode.Compress)
  compStream.Write(fileBytes, 0, fileBytes.Length)
  compStream.Flush()
  compStream.Close()
  writer.Close()
   'Dekomprimieren
  stream = New IO.FileStream(zieldatei, FileMode.Open)
  Dim decompStream As New Compression.GZipStream(stream, _
    IO.Compression.CompressionMode.Decompress)
  Dim reader As New StreamReader (CType(decompStream, Stream))
  File.WriteAllText(My.Computer.FileSystem. _
    SpecialDirectories.MyDocuments & _
    "\dekomprimierte.txt", reader.ReadToEnd())
   reader.Close()
   stream.Close()
End Sub
```

Abbildung 12.2 zeigt den Inhalt der komprimierten Datei.

Abbildung 12.2 Inhalt der komprimierten Datei



12.2.3 Daten ver- und entschlüsseln

Neben der Möglichkeit, Daten zu komprimieren und zu dekomprimieren, bietet das Framework auch noch die Möglichkeit, Daten zu verschlüsseln und zu entschlüsseln. Dieses erfolgt ähnlich wie beim Komprimieren mit einem `Stream`. Die Funktionen zum Entschlüsseln werden in der Klasse `System.Security.Cryptography` zur Verfügung gestellt. Das Beispiel in Listing 12.8 zeigt, wie ein String zunächst verschlüsselt in eine Datei geschrieben und danach wieder entschlüsselt und in der Konsole ausgegeben wird.

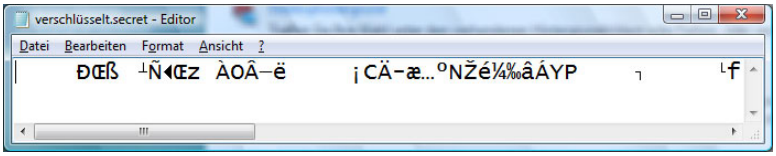
Um diese Funktionalität nutzen zu können, müssen Sie noch einen Verweis auf die Assembly `System.Security` zu Ihrem Projekt hinzufügen.

Listing 12.8
Daten ver- und entschlüsseln

```
Public Sub verschlüsseln()
  Dim text As String = "Lesen macht viel Spaß"

  Dim dStream As MemoryStream
  Dim verschluesselteDaten() As Byte

   ' MemoryStream erzeugen
   dStream = New MemoryStream()
   Dim w As New StreamWriter(dStream)
   w.Write(text)
   w.Close()

   ' String verschlüsseln
   verschluesselteDaten = System.Security.CrypTography. _
     ProtectedData.Protect(dStream.ToArray(), _
     Nothing, System.Security.Cryptography. _
     DataProtectionScope.CurrentUser)

   ' Datei speichern
   File.WriteAllBytes("verschlüsselt.secret", _
     verschluesselteDaten)

End Sub

Public Sub entschluesseln()

  Dim entschluesselteDaten() As Byte
  Dim daten() As Byte

  'Entschlüsselung der Daten
  daten = File.ReadAllBytes("verschlüsselt.secret")
  entschluesselteDaten = _
    System.Security.Cryptography.ProtectedData. _
    Unprotect(Daten, Nothing, _
    System.Security.Cryptography. _
    DataProtectionScope.CurrentUser)

   'Daten wieder auslesen
   Dim dataStream As New MemoryStream(entschluesselteDaten)
   Dim r As New StreamReader(dataStream)
   Console.WriteLine("Daten wurden entschlüsselt: " & _
      r.ReadToEnd())
   r.Close()
End Sub
```

Auch hier will ich Ihnen in Abbildung 12.3 die verschlüsselten Daten in der Datei zeigen.

Abbildung 12.3
Inhalt der verschlüsselten Datei



12.2.4 Dateiüberwachung mit FileSystemWatcher

Die `FileSystemWatcher`-Klasse überwacht Veränderungen im Dateisystem. Dabei können Sie angeben, welcher Ordner überwacht und welche Filtereinstellungen hierfür verwendet werden sollen. Sie können zum Beispiel nach Veränderungen in bestimmten Dateien suchen.

Die Methode `aktiviereUeberwachung()` aus Listing 12.9 überwacht den Ordner *Eigene Dateien* bzw. unter Windows Vista und Windows 7 den Ordner *Dokumente* nach neuen Dateien. Innerhalb dieser Methode wird dem Ereignis `Created` des `FileSystemWatcher` ein Ereignishandler hinzugefügt. Dieser Ereignishandler wird aufgerufen, wenn eine neue Datei angelegt wird. Die Methode `fswatcher_Created()` gibt eine entsprechende Benachrichtigung auf der Konsole aus. Außerdem erzeugt sie eine Textdatei, in der die Änderungen festgehalten werden.

Listing 12.9
FileSystemWatcher im Einsatz

```
Private Sub aktiviereUeberwachung()

  Dim pfad As String = My.Computer.FileSystem. _
    SpecialDirectories.MyDocuments
  Dim fswatcher As New FileSystemWatcher(pfad, "*.*")
  AddHandler fswatcher.Created, AddressOf fswatcher_Created
  fswatcher.NotifyFilter = NotifyFilters.FileName
  fswatcher.EnableRaisingEvents = True
End Sub

Private Sub fswatcher_Created(ByVal sender As Object, _
   ByVal e As IO.FileSystemEventArgs)

   Dim text As String = "Neue Datei gefunden: " & e.Name
   Console.WriteLine(text)
   File.AppendAllText(My.Computer.FileSystem. _
     SpecialDirectories.MyDocuments & "\protokoll.txt", text)
End Sub
```

12.3 Zeichnen mit GDI+

GDI steht für **Graphics Device Interface**. Schon am Namen erkennt man, dass es sich hierbei um die Technologie für Grafiken und Zeichnen handelt. Da **GDI+** Bestandteil des .NET Framework ist – die zugehörigen Klassen finden Sie im Namensraum `System.Drawing` –, kann es direkt in .NET-Programmen angesprochen werden, um damit zum Beispiel transparente Fenster, Animationen oder kombinierte Grafikelemente zu erstellen. Zeichenfunktio-

nen und Malmethoden sind in Klassen eingeteilt. Im Mittelpunkt steht das `Graphics`-Objekt, das in diesem Kontext das Zeichnen von Linien, Ellipsen, Rechtecken etc. ermöglicht.

12.3.1 Grundlagen

GDI+ ist definiert innerhalb von `System.Drawing` und den fünf darin enthaltenen Namensräumen:

- System.Drawing.Design
- System.Drawing.Printing
- System.Drawing.Imaging
- System.Drawing.Drawing2D
- System.Drawing.Text

GDI+ besteht dabei grundsätzlich aus drei Komponenten: zweidimensionalen Vektorgrafiken, Bildverarbeitung (Imaging) und Typografie.

System.Drawing.Graphics

An dieser Klasse kommen Sie nicht vorbei, wenn Sie GDI+ in einer Applikation einbauen wollen. Sie stellt sozusagen die Zeichenoberfläche dar und ist somit die Grundlage von GDI+ und für alle Zeichenoperationen verantwortlich.

Im folgenden Minibeispiel in Listing 12.10 soll demonstriert werden, wie man ein einfaches Rechteck mit dem Graphics-Objekt erstellt.

```
Imports System.Drawing
Public Class Form1

  Private Sub Form1_Paint(ByVal sender As Object, _
     ByVal e As PaintEventArgs) Handles Me.Paint

    e.Graphics.DrawRectangle(Pens.Black, 10, 10, 200, 150)
   End Sub
End Class
```

Listing 12.10
Zeichnen eines einfachen Rechtecks in einem Formular

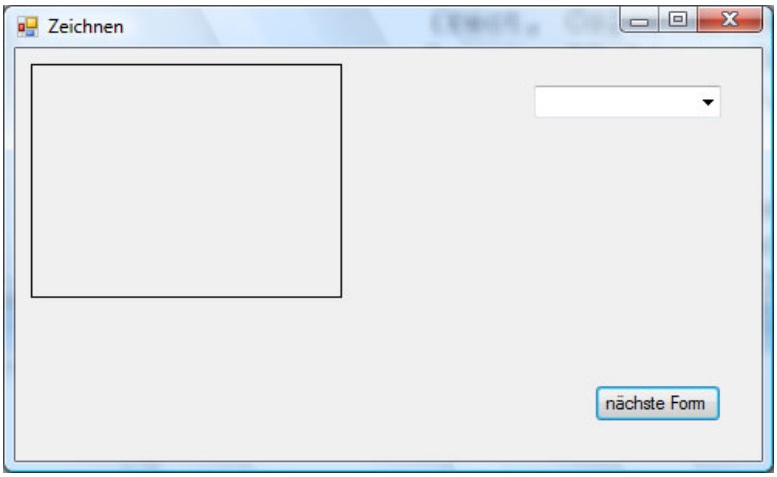


Abbildung 12.4
Das gezeichnete Rechteck

Dabei gibt es mehrere Möglichkeiten, auf das Graphics-Objekt zuzugreifen. Eine Möglichkeit, und zwar die empfohlene, besteht darin, in dem `Paint`-Ereignis der jeweiligen Form die Logik für die Erstellung der Grafik zu implementieren, wie in dem obigen Beispiel geschehen. Dabei verwendet die Methode `DrawRectangle()` einen vorgefertigten Stift, hier `Pens.Black`, und die Koordinaten für das zu zeichnende Rechteck. Das Rechteck hätte auch im `Form1_Load()`-Ereignishandler erzeugt werden können, wie Listing 12.11 zeigt.

Listing 12.11 Zeichnen im Form_Load

```
Private Sub Form1_Load(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load

   Dim g As Graphics = Me.CreateGraphics
   g.DrawRectangle(Pens.Black, 10, 10, 200, 150)
   g.Dispose()
End Sub
```

Nachdem man nun ein `Graphics`-Objekt erzeugt hat oder wie in Listing 12.10 das über die `EventArgs` übergebene `Graphics`-Objekt im `Paint`-Ereignis verwendet, kann man mit dessen Methoden alle möglichen Linien, Kreise oder sonstigen Figuren auf den Bildschirm zeichnen. Der Aufruf von `g.Dispose()` im Listing 12.11 ist bei der Erzeugung eines eigenen `Graphics`-Objekts unbedingt zu empfehlen, da dieses Objekt sehr viele Ressourcen beansprucht, die möglichst schnell wieder freigegeben werden sollten. Bei der Verwendung des `Graphics`-Objekts im `Paint`-Ereignis wird dieses Objekt intern weiterverwendet, deshalb dürfen Sie die `Dispose()`-Methode dort keinesfalls aufrufen.

Auf den ersten Blick scheinen beide Varianten eigentlich gleichwertig, aber es gibt einen wesentlichen Unterschied. Die Methode `Form1_Load()` wird nur beim Laden der Anwendung ausgeführt, während das `Paint`-Ereignis jedes Mal ausgeführt wird, wenn das Formular neu gezeichnet werden soll. Dies kann geschehen, wenn das Formular von einem anderen verdeckt wurde oder wenn es minimiert wurde. Dadurch, dass das Betriebssystem eine Nachricht an Ihre Applikation schickt, die automatisch das `Paint`-Ereignis auslöst, ist das eigentlich die einzige Stelle, an der Sie Zeichenoperationen vornehmen sollten.

Wenn Sie selbst das Neuzeichnen des Formulars anstoßen wollen, müssen Sie lediglich die Methode `Invalidate()` aufrufen.

Achtung

Rufen Sie keinesfalls `Invalidate()` innerhalb des `Paint`-Ereignisses auf, denn das führt zu einer endlosen Kette von rekursiven Aufrufen. Auch Haltepunkte machen innerhalb des `Paint`-Ereignisses keinen Sinn, da der Code und die Entwicklungsumgebung am Bildschirm angezeigt werden und danach das ursprüngliche Formular wieder neu gezeichnet werden muss.

Mit dem `Graphics`-Objekt können folgende geometrische Figuren gezeichnet werden:

- Linien
- Rechtecke
- Bögen

- Ellipsen
- Pies (Tortenstücke)
- Polygone
- Béziers
- Kurven
- Strings

Um nun diese grafischen Konturen definieren und lokalisieren zu können, braucht man noch bestimmte Datentypen.

Point und Size

Zum einen gibt es den `Point` als Äquivalent des 2D-Vektors. Er wird durch zwei Integer-Werte bestimmt, die besagen, wie weit man vom Nullpunkt in horizontaler und dann in vertikaler Richtung gehen muss, um diesen Punkt zu erreichen.

Der Struktur `Size` übergibt man analog zum Punkt zwei Zahlen, mit dem Unterschied, dass hier nicht die X- und die Y-Koordinate gemeint sind, sondern die Breite und die Höhe. Ein Beispiel soll diesen Unterschied verdeutlichen:

Listing 12.12
Rechteckzeichnen mittels Point- und Size-Objekten

```
Dim g As Graphics = e.Graphics
Dim blackpen As New Pen(Color.Black, 4)
Dim ObenLinks As New Point(0, 0)
Dim Flaeche As New Size(100, 100)
Dim Rechteck As New Rectangle(ObenLinks, Flaeche)
g.DrawRectangle(blackpen, Rechteck)
```

Der `Point` *ObenLinks* gibt hier also an, wo die linke obere Ecke des Rechtecks liegt, wobei die *Flaeche* als `Size` die Länge und Breite des Rechtecks festlegt. *Rechteck* stellt in dem obigen Beispiel eine weitere Struktur dar, die man gut für die Definition solcher Grundfiguren gebrauchen kann.

Wenn wir unsere bisherigen Malutensilien nochmals betrachten, dann haben wir jetzt – im übertragenen Sinn – eine Person mit einer Zeichenvorlage in Form der `Graphics`-Klasse und eine Art Koordinatensystem oder auch Lineal, hier die gerade besprochenen Datentypen, mit denen wir unsere Grafikobjekte positionieren und dimensionieren können.

Was wir jetzt noch brauchen, um ein paar Linien auf unsere leere Zeichenebene zu bringen, sind ein paar Stifte, um die Konturen zu zeichnen. Doch da das Ausmalen der Figuren relativ lange dauern würde, haben wir noch einige Pinsel zur Hand.

Wie wir in den bislang gezeigten Codebeispielen gesehen haben, benötigt man für die `Draw()`-Methoden des `Graphics`-Objekts ein **Pen**-Objekt, auf das wir jetzt etwas näher eingehen wollen.

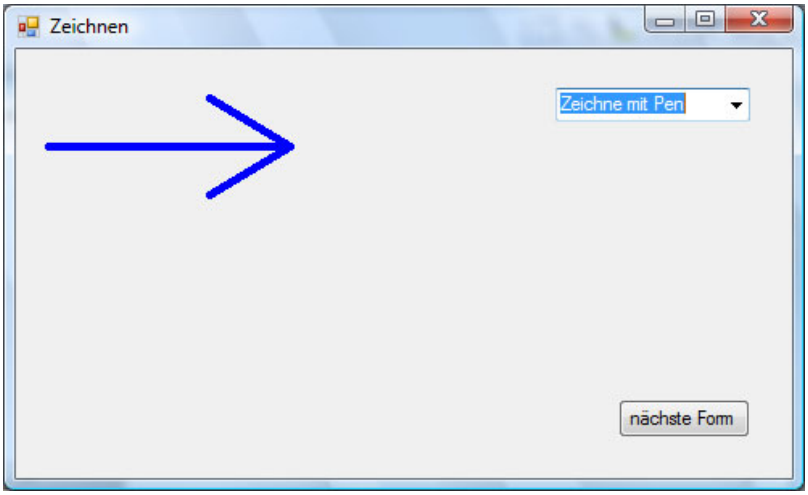
System.Drawing.Pen

Ein Pen zeichnet eine Linie mit speziellen Eigenschaften. Entweder Sie stellen sich selbst einen Pen zusammen, indem Sie die Linienfarbe, Linienbreite und Linienart auswählen (wie in Listing 12.12), oder Sie greifen einfach auf die statische Klasse `Pens` zurück, die Ihnen eine Palette von vorgefertigten Stiften mit unterschiedlicher Farbe anbietet.

Listing 12.13 Zeichnen eines Pfeils mit einem Pen-Objekt

```
Dim g As Graphics = e.Graphics
Dim p As New Pen(Color.Blue, 5)
p.EndCap = Drawing2D.LineCap.Round
p.StartCap = Drawing2D.LineCap.Round
g.DrawLine(p, 20, 60, 170, 60)
g.DrawLine(p, 120, 30, 170, 60)
g.DrawLine(p, 170, 60, 120, 90)
g.Dispose()
```

Abbildung 12.5 Der gezeichnete Pfeil



Die meiner Meinung nach wohl wichtigsten Eigenschaften in diesem Zusammenhang sind

- `Color`: Lesen oder Setzen der Linienfarbe
- `Width`: Lesen oder Setzen der Liniendicke
- `Brush`: Lesen oder Setzen der Brush-Art, welche die Eigenschaften des Pens bestimmt
- `StartCap`: Lesen oder Setzen der Anfangseigenschaften der Linie
- `EndCap`: Lesen oder Setzen der Endeigenschaften der Linie

So wie Sie bei sämtlichen `Draw()`-Methoden einen `Pen` brauchen, so verhält es sich bei den `Fill()`-Methoden mit dem `Brush`.

System.Drawing.Brush

Die `Brush`-Klasse ist eine abstrakte Basisklasse, die zum Füllen von geometrischen Formen dient. Wir können sie also nicht instanziieren und verwenden deshalb entweder eine von uns selbst geschriebene und von dieser abgeleitete Klasse oder eine schon im Framework enthaltene wie etwa `SolidBrush`, `LinearGradientBrush`, `HatchBrush` oder `TextureBrush`.

Dabei ist der `SolidBrush` der einfachste `Brush`, wie der Name schon zu erkennen gibt. Man übergibt ihm eine Farbe beim Konstruktoraufruf, die er dann auch ohne jegliche Veränderung darstellt.

Der `LinearGradientBrush` ist etwas komplexer, wie ich im Listing 12.14 demonstrieren will. Mit dieser Art von `Brush` ist es möglich, Farbverläufe zu zeichnen.

```
Dim g As Graphics = e.Graphics
Dim rect As New Rectangle(20, 20, 100, 100)
Dim brush As New Drawing2D.LinearGradientBrush _
  (rect, Color.Azure, Color.Coral, _
   Drawing2D.LinearGradientMode.BackwardDiagonal)
g.FillRectangle(brush, rect)
```

Listing 12.14
Zeichnen von Farbverläufen mit einem LinearGradientBrush-Objekt

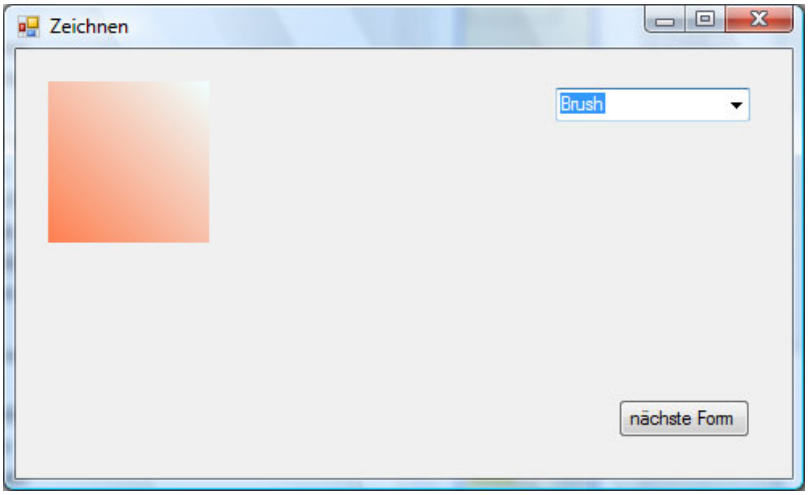


Abbildung 12.6
Rechteck mit einem LinearGradientBrush gezeichnet

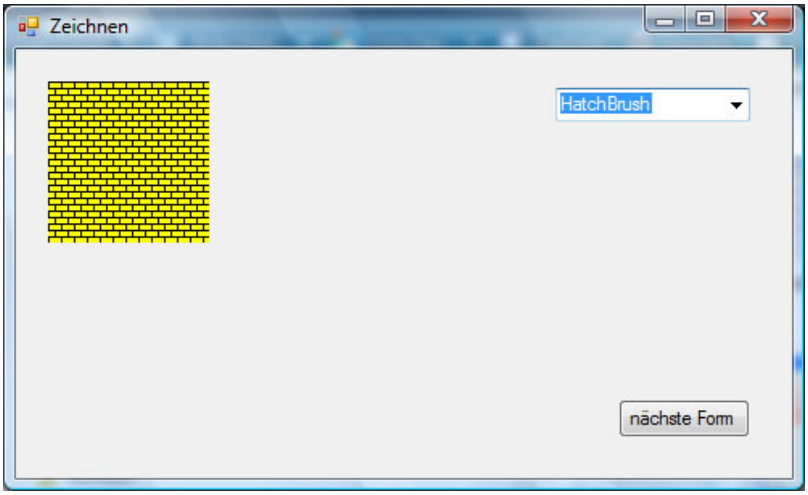
Um mit dem `LinearGradientBrush` eine Fläche zu füllen, benötigt man mindestens ein Rechteck, in dem der Farbverlauf wirksam werden soll, eine Anfangs- und eine Zielfarbe und die Richtung des Farbverlaufs. Das ist lange noch nicht alles, was man mit diesem Pinsel anstellen kann, und eine detaillierte Recherche in Bezug auf die visuellen Möglichkeiten in diesem Zusammenhang, die ich hier nur andeuten möchte, werden Sie mit Sicherheit nicht bereuen.

Etwas ganz anderes kann man mit dem `HatchBrush` erzeugen. Mit diesem kann man ein Wiederholungsmuster von über 50 vordefinierten Mustern (z. B. ein Mauerwerk) abbilden, deren Vorder- und Hintergrundfarbe man selbst bestimmt.

Listing 12.15
Zeichnen mit einem HatchBrush-Objekt

```
Dim g As Graphics = e.Graphics
Dim rect As New Rectangle(20, 20, 100, 100)
Dim brush As New Drawing2D.HatchBrush _
    (Drawing2D.HatchStyle.HorizontalBrick, _
    Color.Black, Color.Yellow)
g.FillRectangle(brush, rect)
```

Abbildung 12.7
Beispiel für das Zeichnen mit einem HatchBrush



In der MSDN finden Sie zu jedem Hatch-Style ein kleines Bild, wodurch man den gewünschten passenden Style sehr schnell findet.

Nun kommen wir zu dem rechenintensivsten Pinsel, mit dem man beliebige Muster darstellen kann, dem `TextureBrush`. Einfach gesagt braucht er nur ein Bild von einem Muster, womit er dann die angegebene Fläche füllt.

Listing 12.16
Zeichnen mit TextureBrush

```
Dim img As Image
img = Image.FromFile(Application.StartupPath & _
   "\prärіewind.bmp")
Dim brush As New TextureBrush(img)
Dim g As Graphics = e.Graphics
g.FillEllipse(brush, 10, 10, 100, 50)
```

Listing 12.17 zeigt den Beispielcode für das Formular. Bei jedem Ändern des Eintrags in der `ComboBox` wird die Methode `Invalidate()` aufgerufen, was zu einem Neuzeichnen des Formulars führt. In Abhängigkeit der Auswahl wird dann in der `Form1_Paint()`-Methode das entsprechende Bild gezeichnet.

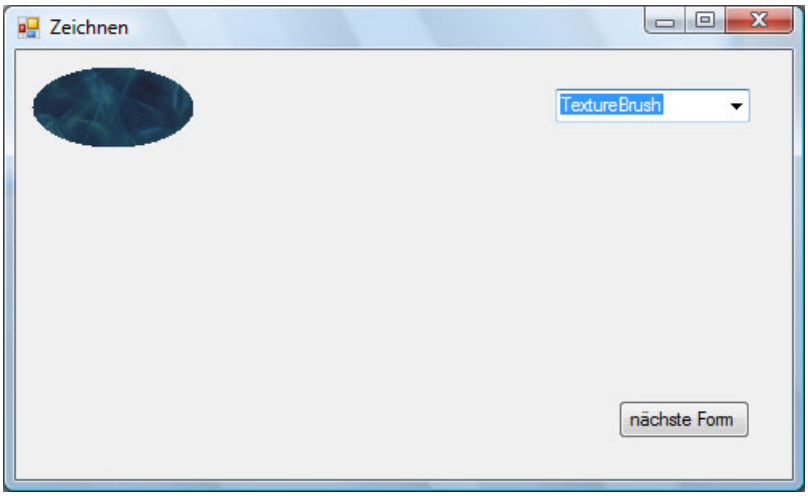


Abbildung 12.8
Beispiel für Zeichnen mit TextureBrush

Listing 12.17
Beispielcode des Formulars

```
Imports System.Drawing
Public Class Form1

  Private Sub Form1_Paint(ByVal sender As Object, _
     ByVal e As PaintEventArgs) Handles Me.Paint

    Dim g As Graphics = e.Graphics
    Dim rect As New Rectangle(20, 20, 100, 100)

      Select Case ComboBox1.Text
        Case "Brush"
          Dim brush As New Drawing2D.LinearGradientBrush _
            (rect, Color.Azure, Color.Coral, _
             Drawing2D.LinearGradientMode.BackwardDiagonal)
          g.FillRectangle(brush, rect)

        Case "HatchBrush"
          Dim brush As New Drawing2D.HatchBrush _
            (Drawing2D.HatchStyle.HorizontalBrick, _
              Color.Black, Color.Yellow)
          g.FillRectangle(brush, rect)

        Case "TextureBrush"
          Dim img As Image
            img = Image.FromFile(Application.StartupPath & _
               "\präriewind.bmp")
          Dim brush As New TextureBrush(img)
          g.FillEllipse(brush, 10, 10, 100, 50)

        Case "Zeichne mit Pen"
          Dim p As New Pen(Color.Blue, 5)
          p.EndCap = Drawing2D.LineCap.Round
          p.StartCap = Drawing2D.LineCap.Round
          g.DrawLine(p, 20, 60, 170, 60)
          g.DrawLine(p, 120, 30, 170, 60)
          g.DrawLine(p, 170, 60, 120, 90)
```

Listing 12.17 (Forts.) Beispielcode des Formulars

```
        Case "Zeichne"
          Dim blackpen As New Pen(Color.Black, 4)
          Dim ObenLinks As New Point(0, 0)
          Dim Flaeche As New Size(100, 100)
          Dim Rechteck As New Rectangle(ObenLinks, Flaeche)
          g.DrawRectangle(blackpen, Rechteck)

        Case Else
          g.DrawRectangle(Pens.Black, 10, 10, 200, 150)
      End Select
  End Sub

  Private Sub ComboBox1_SelectedIndexChanged _
    (ByVal sender As System.Object, _
     ByVal e As System.EventArgs)
     Handles ComboBox1.SelectedIndexChanged
        Invalidate()
  End Sub
  Private Sub Button1_Click(ByVal sender As System.Object, _
     ByVal e As System.EventArgs) Handles Button1.Click
     Dim d As New Drawing2
     d.Show()
  End Sub
End Class
```

12.3.2 Die wichtigsten Formen und Linien

Jetzt, da wir uns einen `Pen` definieren können, also Linien zeichnen und die Innenfläche mit einem unserer vielfältigen `Brushes` befüllen, kehren wir noch einmal zur `Graphics`-Klasse zurück. `System.Drawing.Graphics` besitzt eine sehr große Anzahl an Methoden, so dass es fast schwierig wird, den Überblick darüber zu behalten. Folgende Liste soll dabei helfen, die wichtigsten `Shapes` und `Lines` im Überblick zu behalten, wobei ich noch explizit erwähnen möchte, dass dies nur ein Ausschnitt aus den möglichen `Draw`- oder `Fill`-Methoden ist:

- DrawLine()
- DrawRectangle()
- DrawEllipse()
- DrawLines()
- DrawCurve()
- DrawArc()
- DrawCloseCurve()
- DrawPie()
- DrawPolygon()
- FillPie()
- FillRectangle()
- FillEllipse()

Die Methode `DrawLine()` verbindet ganz einfach zwei Punkte miteinander.

```
g.DrawLine(Pens.Black,50,50,100,50)
```

Während die Methode `DrawLines()` ein Array von Punkten miteinander verbindet.

```
Dim g As Graphics = e.Graphics
Dim p As New Pen(Color.Blue, 5)
Dim points As Point() = {New Point(50, 40), _
    New Point(70, 50), New Point(80, 30)}
g.DrawLines(p, points)
```

Listing 12.18
Beispiel für DrawLines

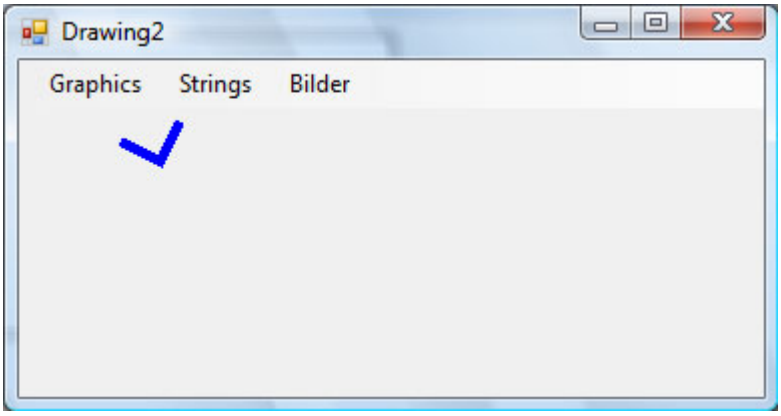


Abbildung 12.9
DrawLines

Mittels der Methoden `DrawRectangle()` oder `FillRectangle()` kann man (befüllte) Rechtecke zeichnen, wie ich es in den Anfangsbeispielen bereits gezeigt habe.

Die Methode `DrawEllipse()` oder `FillEllipse()` zeichnet beziehungsweise füllt eine Ellipse, die durch ein gebundenes Rechteck definiert wird.

```
Dim g As Graphics = e.Graphics
g.FillEllipse(Brushes.Blue, 50, 50, 100, 80)
'Zur Veranschaulichung das gebundene Rechteck
g.DrawRectangle(Pens.Red, 50, 50, 100, 80)
```

Listing 12.19
Beispiel für FillEllipse()

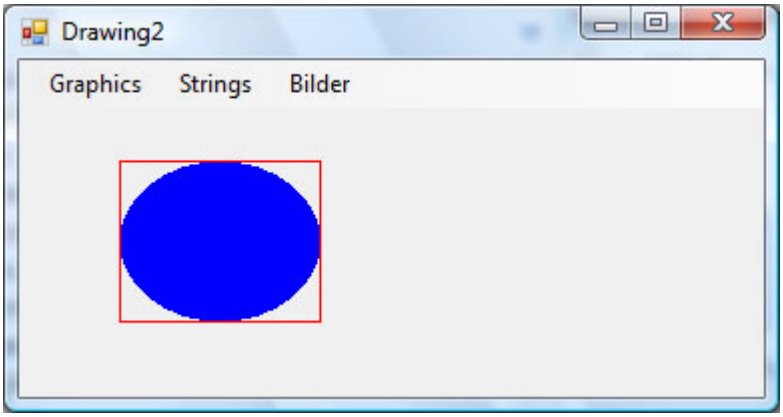


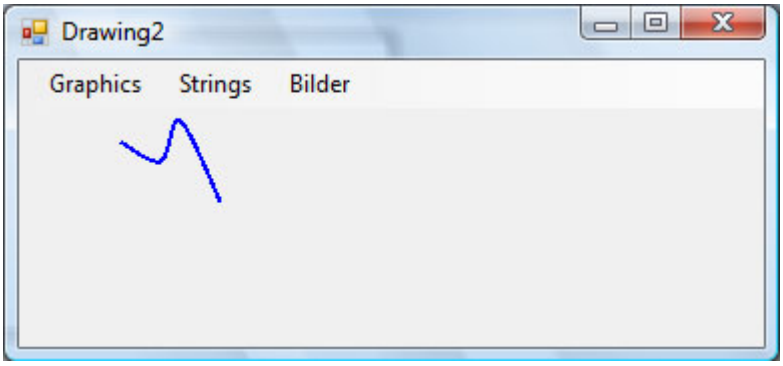
Abbildung 12.10
Ellipse mit gebundenem Rechteck

Wenn Sie Kurven zeichnen wollen, dann können Sie die Methode `DrawCurve()` verwenden, wobei mindestens ein `Pen` und ein Punkte-Array übergeben werden müssen.

Listing 12.20
Beispiel für DrawCurve()

```
Dim g As Graphics = e.Graphics
Dim points As Point() = {New Point(50, 40), _
    New Point(70, 50), New Point(80, 30), _
    New Point(100, 70)}
Dim p As New Pen(Color.Blue, 2)
g.DrawCurve(p, points)
```

Abbildung 12.11
Gezeichnete Kurven

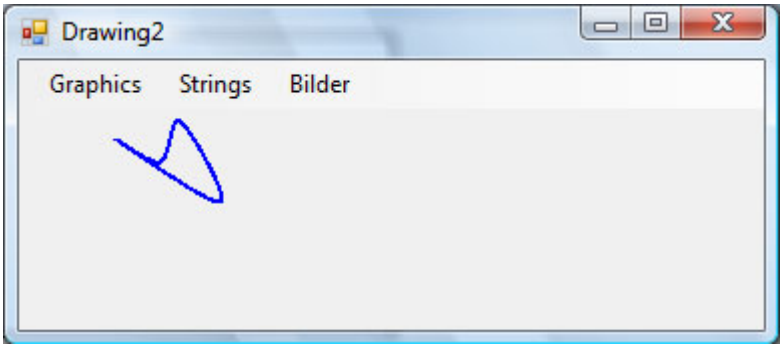


Für diese Methode gibt es auch noch mehrere Überladungen (unter anderem mit fünf Übergabeparametern), bei denen man zusätzlich noch angeben kann, ab welchem Punkt zu zeichnen begonnen werden soll und wie stark die Krümmung der Kurve sein soll.

Die Methode `DrawClosedCurve()` funktioniert genauso wie `DrawCurve()`, mit dem Unterschied, dass der letzte mit dem ersten Punkt verbunden wird.

```
g.DrawClosedCurve(p, points)
```

Abbildung 12.12
Geschlossene Kurve



Die `DrawArc()`-Methode ist da schon ein wenig komplizierter. Sie zeichnet einen Bogen, der einer Ellipse entnommen wurde.

Listing 12.21
Beispiel für DrawArc()

```
Dim g As Graphics = e.Graphics
Dim p As New Pen(Color.Blue, 2)
g.DrawArc(p, 30, 30, 200, 150, 90, 180)
```

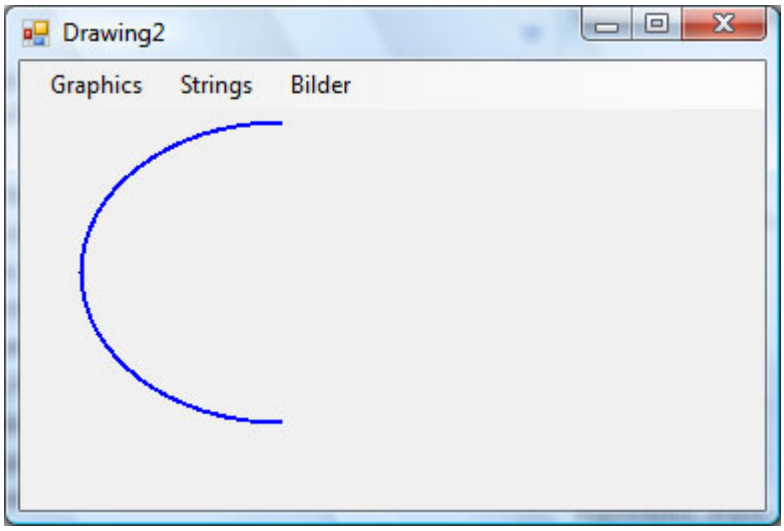


Abbildung 12.13
Halbbogen

Die ersten vier Parameter (30, 30, 200, 150) bilden ein gebundenes Rechteck als Basis für eine Ellipse (siehe `DrawEllipse()`). Der gezeichnete Bogen ist ein Ausschnitt dieser Ellipse, dessen umrahmendes Rechteck wir angegeben haben. Die `Graphics`-Klasse nimmt nun den Mittelpunkt dieser Ellipse und geht von diesem in unserem Beispiel 90° (sechster Paramter) im Uhrzeigersinn, um den Startpunkt des zu zeichnenden Bogens zu finden. Der letzte Parameter (hier 180) sagt der Methode nun, um wie viel Grad von dem jetzigen Startpunkt im Uhrzeigersinn ausgehend man weitergehen soll, um den Endpunkt des Bogens zu erhalten.

Kommen wir beim vorletzten Beispiel noch zum Tortenmalen. Vom Prinzip her funktioniert diese Malmethode genauso wie die gerade eben besprochene `DrawArc()`-Methode. Um solch eine Torte nun auf den Bildschirm zu bringen, wenn Sie zum Beispiel irgendwelche Produktivdaten aus dem Betrieb dynamisch in solchen Diagrammen darstellen möchten, brauchen Sie wie vorhin ein Rechteck, das wiederum eine Ellipse umgibt, die wir uns als Tortenform vorstellen können. Die Torte/Pie wird anschließend gezeichnet, und zwar ab dem von uns angegebenen Startgrad bis zu dem Endgrad, gemessen ab dem Startgrad.

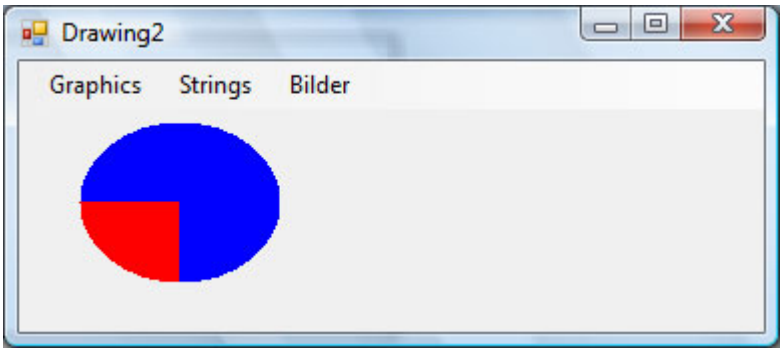
Listing 12.22
Beispiel für FillPie()

```
Dim g As Graphics = e.Graphics
g.FillPie(Brushes.Blue, 30, 30, 100, 80, 180, 270)
g.FillPie(Brushes.Red, 30, 30, 100, 80, 90, 90)
```

Tipp

Um den Blickwinkel nun etwas perspektivischer von der Seite zu gestalten, müssen Sie einfach die hinterlegte Ellipsenform ändern. Je weniger die Ellipse einem Kreis ähnelt, desto mehr wandert der Blick von oben zur Seite.

Abbildung 12.14
Tortendarstellung

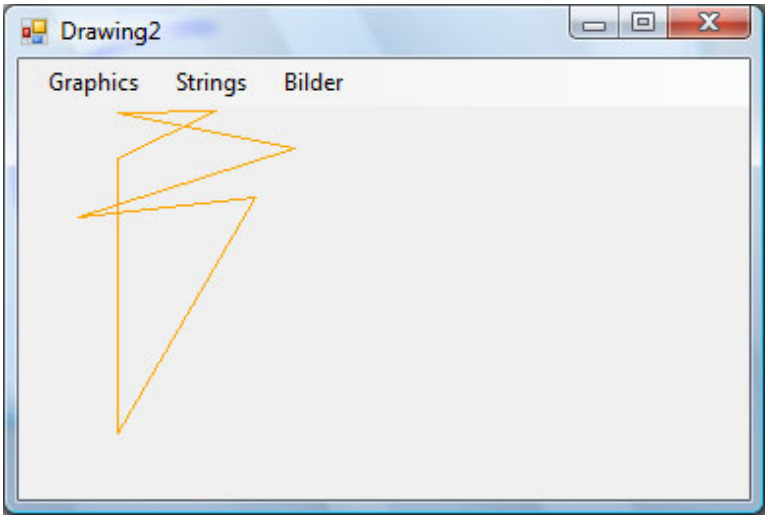


Zum Abschluss wollen wir noch ein Polygon zeichnen. Ein Polygon (aus dem Griechischen: polys = viel und gonos = Winkel) ist ein Begriff aus der Geometrie. Dieses erhält man, wenn man mindestens drei voneinander verschiedene Punkte in einer Zeichenebene durch Strecken so miteinander verbindet, dass eine geschlossene Figur entsteht, wobei Dreiecke, Vierecke und Sechsecke Beispiele für besondere Polygone sind. Um solch ein Gebilde zu zeichnen, benötigen wir nur einen Pen und ein Array aus Punkten. Intern wird dann ein Punkt nach dem anderen mit einer Linie verbunden, bis sich der Kreis wieder schließt und wir am Anfang sind.

Listing 12.23
Beispiel für DrawPolygon()

```
Dim g As Graphics = e.Graphics
Dim p1 As New Point(50, 50)
Dim p2 As New Point(100, 26)
Dim p3 As New Point(50, 27)
Dim p4 As New Point(140, 45)
Dim p5 As New Point(30, 80)
Dim p6 As New Point(120, 70)
Dim p7 As New Point(50, 190)
Dim points As Point() = {p1, p2, p3, p4, p5, p6, p7}
g.DrawPolygon(Pens.Orange, points)
```

Abbildung 12.15
Polygon



12.3.3 GraphicsPath

Die Klasse `GraphicsPath`, im Namespace `System.Drawing.Drawing2D`, ist eine Art Container für grafische Grundfiguren. Diesem Objekt kann man folgende Grundfiguren mittels der Methode `Add()` übergeben:

- Linien, Rechtecke, Bögen, Ellipsen
- Polygone, Bézier, Splines (Kurven)
- Strings
- »Tortenstücke« (Pie)
- GraphicsPaths

Um nun ein `GraphicsPath`-Objekt darzustellen, gehen Sie folgendermaßen vor:

1. Klasse `GraphicsPath` instanziieren:
2. `StartFigure()` aufrufen
3. Geplante Figuren mittels `Add()` hinzufügen
4. `CloseFigure()` aufrufen
5. Wenn noch mehrere Objekte folgen, bei Schritt 2 wieder beginnen
6. Mit einer Instanz von `Graphics` die Methode `DrawPath()` aufrufen

Hier ein kleines Beispiel, in dem eine Linie, ein Rechteck und eine Ellipse gezeichnet werden.

Listing 12.24
Beispiel für DrawPath()

```
Dim g As Graphics = e.Graphics
Dim gPath As New Drawing2D.GraphicsPath
Dim start As New Point(20, 50)
Dim ende As New Point(100, 50)
gPath.StartFigure()
gPath.AddLine(start, ende)
gPath.AddRectangle(New Rectangle(10, 30, 50, 80))
gPath.AddEllipse(10, 30, 50, 80)
gPath.CloseFigure()
g.DrawPath(Pens.Blue, gPath)
```

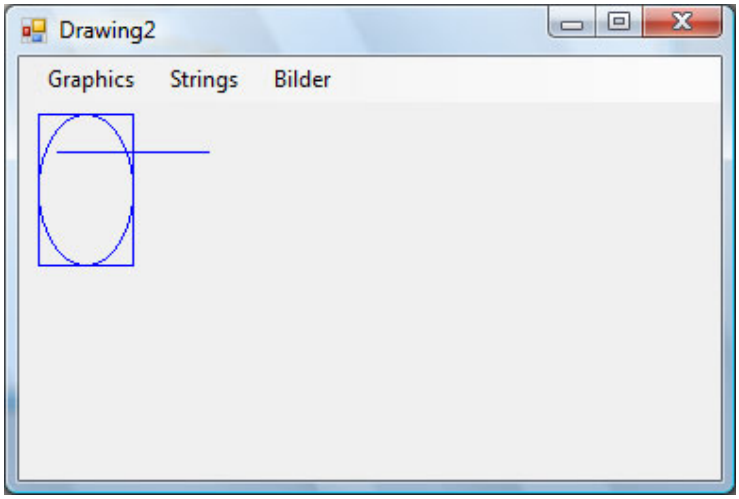


Abbildung 12.16
GraphicsPath-Objekt

Die Vorteile dieser Klasse liegen natürlich auf der Hand. Alle Formen können mit einem Aufruf gezeichnet, transformiert und durch Aufruf der Methode `Reset()` wieder vom Bildschirm gelöscht werden. Informationen über die in der Instanz enthaltenen Punkte und Figuren können über die Properties `PointCount`, `PathPoints` und `PathTypes` in Erfahrung gebracht werden.

Möchte man bei einer komplexen grafischen Figur wissen, ob der User mit der Maus in die Figur hineingeklickt hat oder ob vielleicht auf den Rand geklickt wurde, so kann man das hier sehr leicht testen, indem man die Methoden `IsVisible()` und `IsOutlineVisible()` einer Instanz der Klasse `GraphicsPath` benutzt. Geben Sie dabei die Koordinaten eines Punkts an und der Rückgabewert dieser Methode sagt Ihnen, ob sich dieser Punkt innerhalb – bei `IsOutlineVisible()` auch auf der Linie – des `GraphicPath`-Objekts befindet.

12.3.4 Transformationen

Transformationen nehmen in der Welt der Grafiker einen sehr großen Stellenwert ein und auch im Bereich der GDI+-Welt werden sie wohl immer öfter zum Einsatz kommen und sich größerer Beliebtheit erfreuen. Das .NET Framework ermöglicht es auch Programmierern mit geringen Grundkenntnissen der Mathematik, erstaunliche Dinge zu programmieren.

Grundsätzlich gibt es vier Möglichkeiten, ein Objekt am Bildschirm zu transformieren. Man kann es drehen (**Rotation**), verschieben (**Translation**), vergrößern und verkleinern (**Scaling**) und Scherungen (**Shearing**) durchführen.

Diese Transformationen führt nun in der GDI+-Welt die Klasse `Matrix` für uns aus. Sie befindet sich im Namensraum `System.Drawing.Drawing2D` und gibt uns mit den Methoden `Rotate()`, `Scale()`, `Translate()` und `Shear()` ein mächtiges Werkzeug in die Hand, dessen Möglichkeiten hier kurz dargestellt werden sollen.

Betrachten wir folgenden Code-Ausschnitt in Listing 12.25.

Listing 12.25 Beispiel für Transformation

```
Dim rect As New Rectangle(100, 100, 50, 50)
Dim g As Graphics = e.Graphics
Dim m As New Drawing2D.Matrix()

g.DrawRectangle(Pens.Black, rect)
For i As Integer = 0 To 10
  m.Translate(-100, -100, Drawing2D.MatrixOrder.Append)
  m.Rotate(30, Drawing2D.MatrixOrder.Append)
  m.Translate(100, 100, Drawing2D.MatrixOrder.Append)
  g.Transform = m
  g.DrawRectangle(Pens.Black, rect)
Next
```

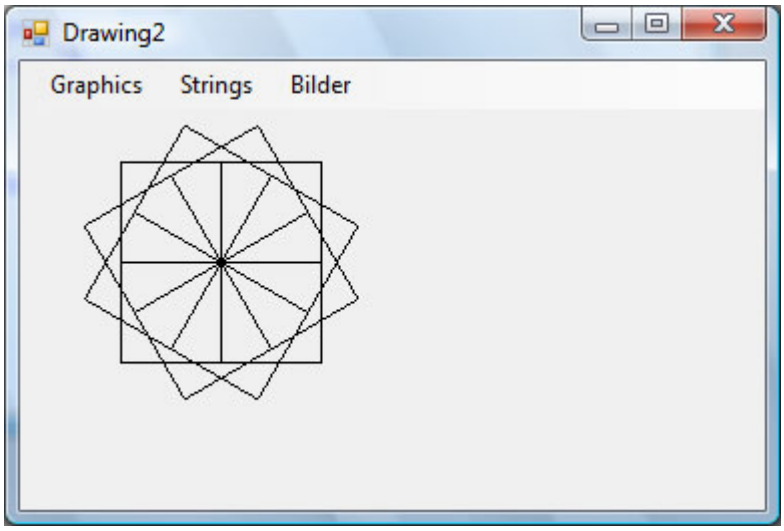


Abbildung 12.17
Transformation eines Rechtecks

Hier wird ein Rechteck – in diesem Fall ein Quadrat – am Punkt (100;100) mit einer Breite und Höhe von 50 gezeichnet. Anschließend wird es jeweils um 30° im Uhrzeigersinn an der linken oberen Ecke elfmal gedreht, bis es schließlich wieder seinen Ausgangspunkt erreicht hat. Und all das macht die `Graphics`-Klasse für uns, nachdem wir ihrer Eigenschaft `Transform` unsere geschaffene `Matrix` übergeben haben. In der `For`-Schleife bewegen wir das Rechteck zuerst mit `Translate()` zur linken oberen Ecke der Form, drehen es schließlich um 30° nach rechts und schieben es wieder an seine Ursprungsposition zurück. Das machen wir deshalb, da sich der Drehpunkt standardmäßig an der Koordinate 0;0 befindet.

Mit der Methode `RotateAt()` hätten Sie den Drehpunkt auch direkt angeben können und sich die beiden `Translate()`-Anweisungen sparen können.

`MatrixOrder.Append` legt schließlich in Bezug auf die Reihenfolge bei der Matrixmultiplikation fest, dass die Transformationsmatrix mit der Matrix multipliziert wird und nicht umgekehrt, da bei Matrizenmultiplikation im Gegensatz zur Zahlenmultiplikation die Reihenfolge beachtet werden muss.

12.3.5 Textdarstellungen

`DrawString()` – das ist die Methode, mit der Sie Text auf der Oberfläche erscheinen lassen können. Doch um einen Text auf den Monitor zu zaubern, müssen wir uns erst ein paar Gedanken darüber machen, welche Schriftart und Schriftgröße wir verwenden möchten. Natürlich werden wir dabei wieder von unseren GDI+-Namensräumen unterstützt, die uns die passenden Objekte zur Verfügung stellen.

Sie haben sich also einen Text ausgedacht und wollen diesen auf der Form an einer bestimmten Stelle mit einer festgelegten Größe ausgeben lassen. Dieses Vorhaben können wir mit nur zwei Zeilen Code bewerkstelligen:

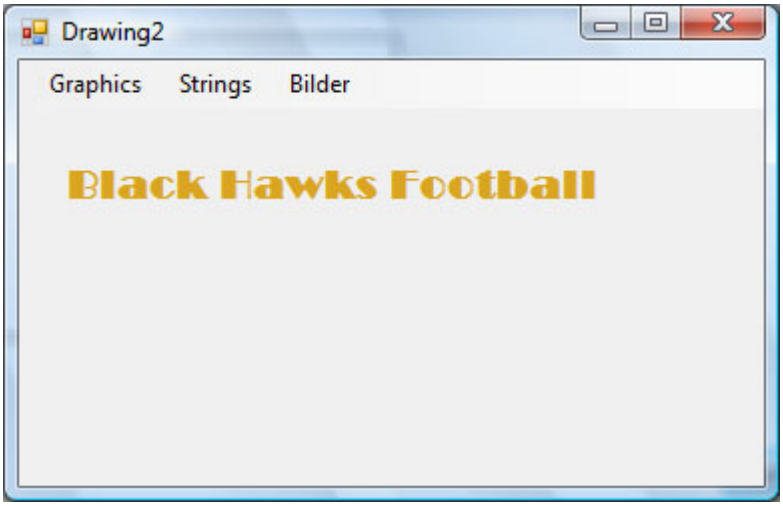
Listing 12.26
Beispiel für Textausgabe

```
Dim g As Graphics = e.Graphics()
g.DrawString("Black Hawks Football", _
    New Font(New FontFamily("Broadway"), 16, _
    FontStyle.Bold), Brushes.Black, 20, 50)
```

Einfach, oder? Wir brauchen also nur die `DrawString()`-Methode einer `Graphics`-Instanz aufzurufen, wobei wir für den zu schreibenden Text die Schriftart, Größe, Style, Farbe und Ausgangspunkt übergeben. Dabei benutzt GDI+ einen von fünf Font-Styles, um die Schrift nach unserem Wunsch darzustellen: `Regular`, `Bold`, `Italic`, `Strikeout` und `Underline`.

Die Textausgabe im Formular sieht dann aus wie in Abbildung 12.18 dargestellt.

Abbildung 12.18
Einfache Textausgabe



Doch was kann man machen, wenn Sie den Text nur innerhalb einer bestimmten Fläche platzieren dürfen, weil die rechte Seite für andere Controls reserviert ist? Für ein solches Szenario bietet uns die `DrawString()`-Methode eine Überladung an, die den Text in ein von uns übergebenes Rechteck schreibt.

Listing 12.27
Beispiel für positionierte Textausgabe

```
Dim g As Graphics = e.Graphics
Dim rect As New Rectangle(20, 50, 130, 80)
g.DrawRectangle(Pens.Black, rect)
g.DrawString("Black Hawks Football", _
    New Font(New FontFamily("Broadway"), 16, _
    FontStyle.Bold), Brushes.Goldenrod, rect)
```

Die Ausgabe im Formular sehen wir in Abbildung 12.19.

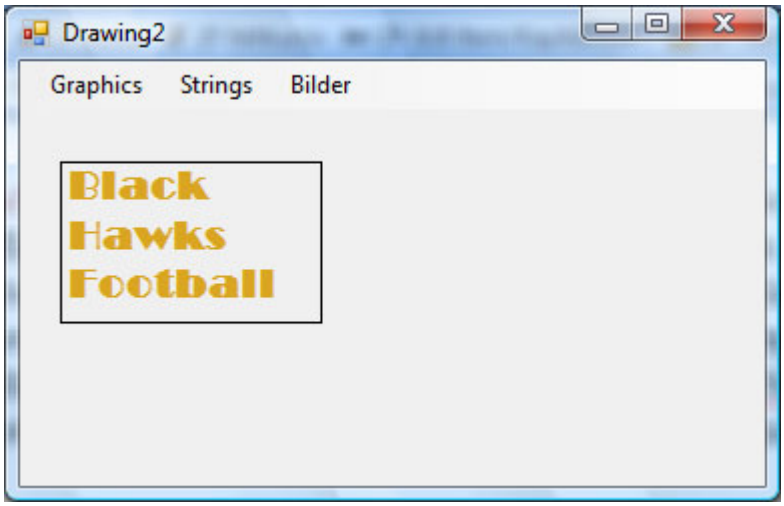


Abbildung 12.19
Positionierte Textausgabe

Statt der X- und Y-Koordinate als Parameter für den Ausgangspunkt übergeben wir jetzt nur ein Rechteck, in dem der Text dargestellt werden soll. Zur Veranschaulichung habe ich das Rechteck auch mit einem schwarzen Rand gezeichnet, normalerweise werden Sie das Rechteck wohl nicht mitzeichnen.

Um den Text nun noch z.B. rechtsbündig auszurichten und in vertikaler Richtung zu zentrieren, wie in Abbildung 12.20 dargestellt, bedienen wir uns des `StringFormat`-Objekts aus dem Namensraum `System.Drawing`. Dessen Eigenschaften `Alignment` und `LineAlignment` müssen wie folgt belegt werden, bevor wir es wie gewohnt unserer `DrawString()`-Methode als letzten Parameter übergeben:

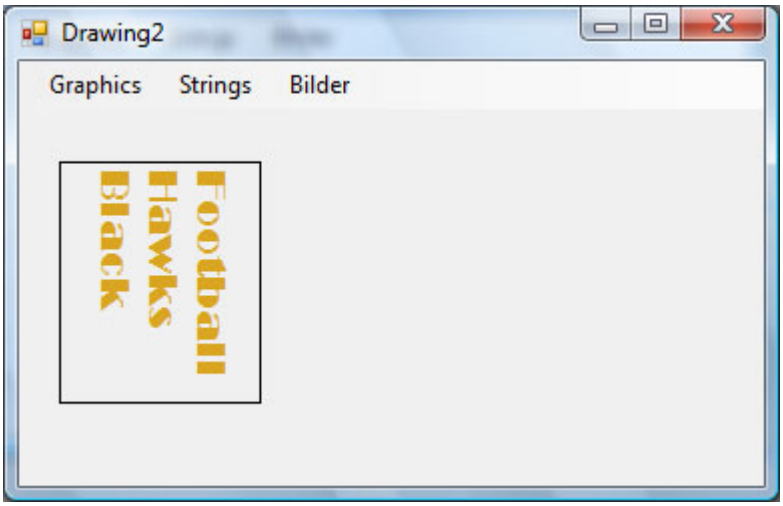
Doch die `StringFormat`-Klasse kann noch mehr. Sie ist nicht nur in der Lage, einen Text vertikal auszurichten, sondern auch vertikal zu schreiben. Dazu geben Sie der Eigenschaft `FormatFlags` den Wert `StringFormatFlags.DirectionVertical` und schon wird der Text vertikal dargestellt. Wobei das nur eine der vielfältigen Möglichkeiten der `FormatFlags` ist.

Listing 12.28
Beispiel für DrawString() mit Textausrichtungsformat

```
Dim g As Graphics = e.Graphics
Dim rect As New Rectangle(20, 50, 100, 120)
g.DrawRectangle(Pens.Black, rect)
Dim f As New StringFormat()
f.Alignment = StringAlignment.Near
f.LineAlignment = StringAlignment.Center
f.FormatFlags = StringFormatFlags.DirectionVertical
g.DrawString("Black Hawks Football", _
  New Font(New FontFamily("Broadway"), 16, _
  FontStyle.Bold), Brushes.Goldenrod, rect, f)
```

Und in Abbildung 12.20 sehen Sie das entsprechende Ergebnis.

Abbildung 12.20
DrawString() mit StringFormat-Optionen



Andere Aufgabenstellung: Sie möchten den Text in Tabellenform mit einer speziellen Spaltenbreite ausgeben. Nun ist es aber nicht so leicht, wie in Word eine Tabelle einzufügen und die einzelnen Zellen mit Buchstaben zu füllen. Wie würden Sie dies zum Beispiel in einer Textdatei realisieren? Genau, mit Tab-Stopps und Zeilenumbrüchen. Und auf diese Weise funktioniert es auch in der GDI+-Welt, wie Sie in Listing 12.29 sehen.

Listing 12.29
Beispiel für Textausgabe mit Tab-Stopps

```
Dim g As Graphics = e.Graphics
Dim tabs() As Single = {120.0, 100.0}
Dim text As String = "Spieltag" & vbTab & "Uhrzeit" & _
  vbTab & "Gegner" & vbCrLf & vbCrLf
text &= "08.05.2010" & vbTab & "15:00" & _
  vbTab & "Lions" & vbCrLf
text &= "26.06.2010" & vbTab & "15:00" & _
  vbTab & "Cowboys" & vbCrLf
Dim f As New StringFormat
f.SetTabStops(0, tabs)
g.DrawString(text, New Font(New FontFamily("Arial"), 16, _
FontStyle.Regular), Brushes.Black, 20, 50, f)
```

Um die Tabulatorenabstände zu setzen, rufen Sie die `SetTabStops()`-Methode eines `StringFormat`-Objekts auf und übergeben dieser zuerst eine Zahl vom Typ `Single`, die den Abstand zwischen dem Beginn einer Zeile und des ersten Tab-Stopps festlegt, und anschließend ein Single-Array, mit dem die Abstände zwischen den einzelnen Tab-Stopps bestimmt werden.

Tipp

Übergeben Sie ein Array, das nur eine Zahl beinhaltet, so erhalten alle Tab-Abstände diesen Wert.

Die entsprechende Ausgabe ist in Abbildung 12.21 dargestellt.

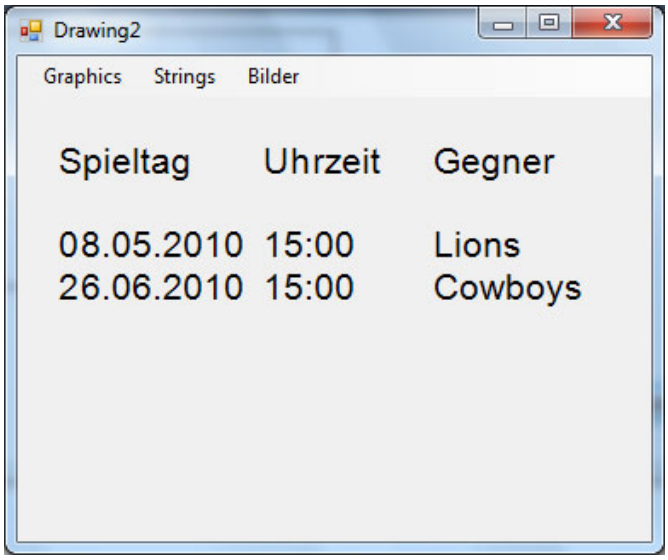


Abbildung 12.21
Textausgabe mit Tab-Stopps

Zum Abschluss der Betrachtung von Textdarstellungen noch eine kleine Spielerei. Wir wollen einen Farbverlauf mittels eines `LinearGradientBrush` erzeugen und der `DrawString()`-Methode übergeben.

Dies demonstriert das Listing 12.30:

Listing 12.30
Beispiel für DrawString() mit Farbverläufen

```
Dim g As Graphics = e.Graphics
Dim tabs() As Single = {120.0, 100.0}
Dim brush As New Drawing2D.LinearGradientBrush _
  (New Rectangle(20, 50, 100, 200), _
  Color.Black, Color.Goldenrod, _
  Drawing2D.LinearGradientMode.Horizontal)
Dim text As String = _
  "Spieltag" & vbTab & "Uhrzeit" & _
  vbTab & "Gegner" & vbCrLf & vbCrLf
text &= "08.05.2010" & vbTab & "15:00" & _
  vbTab & "Lions" & vbCrLf
text &= "26.06.2010" & vbTab & "15:00" & _
  vbTab & "Cowboys" & vbCrLf
Dim f As New StringFormat
f.SetTabStops(0, tabs)
g.DrawString(text, New Font(New FontFamily("Arial"), 16, _
  FontStyle.Regular), brush, 20, 50, f)
```

In Abbildung 12.22 sehen wir das Ergebnis.

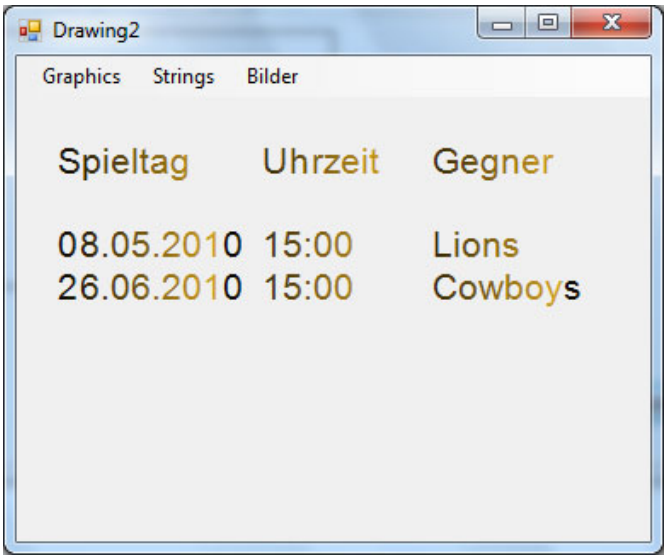


Abbildung 12.22 DrawString() mit Farbverläufen

Man sieht, dass der Farbverlauf hier mehrfach angewandt werden musste, da das Rechteck, in dem sich der Farbverlauf abspielt, für die Darstellung des Textes nicht genügend Platz zur Verfügung gestellt hat. Dies geschieht jedoch völlig automatisch, ohne dass wir das explizit berechnen müssten.

Mit der Methode `MeasureString()` eines `Graphics`-Objekts könnten Sie die Breite des Textes berechnen lassen.

Listing 12.31 zeigt die Darstellung eines exakten Farbverlaufs.

Listing 12.31 Beispiel für DrawString() mit exaktem Farbverlauf

```
Dim g As Graphics = e.Graphics
Dim fo As New Font(New FontFamily("Arial"), 16, _
    FontStyle.Regular)
Dim schriftSize As SizeF = g.MeasureString("Cowboys", fo)
Dim breite As Integer = 120 + 100 + schriftSize.Width

Dim brush As New Drawing2D.LinearGradientBrush _
  (New Rectangle(20, 50, breite, 200), _
   Color.Black, Color.Goldenrod, _
   Drawing2D.LinearGradientMode.Horizontal)
Dim tabs() As Single = {120.0, 100.0}
Dim text As String = "Spieltag" & vbTab & "Uhrzeit" & _
  vbTab & "Gegner" & vbCrLf & vbCrLf
text &= "08.05.2010" & vbTab & "15:00" & _
  vbTab & "Lions" & vbCrLf
text &= "26.06.2010" & vbTab & "15:00" & _
  vbTab & "Cowboys" & vbCrLf

Dim f As New StringFormat
f.SetTabStops(0, tabs)
g.DrawString(text, fo, brush, 20, 50, f)
```

Der exakte Farbverlauf wird in Abbildung 12.23 dargestellt.

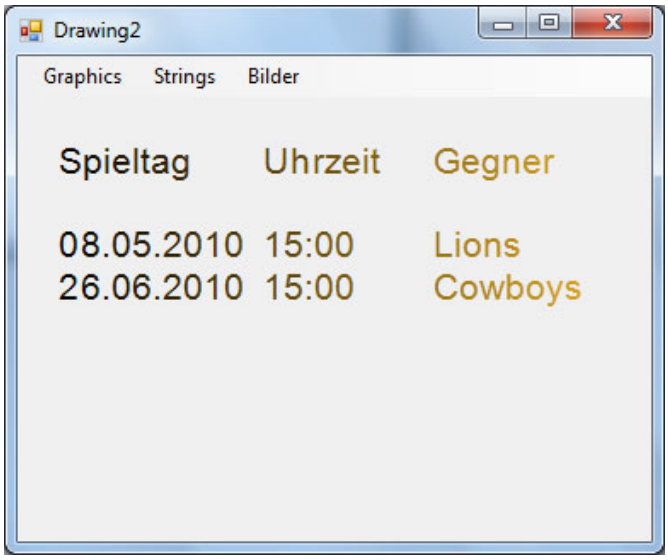


Abbildung 12.23
DrawString() mit exaktem Farbverlauf

12.3.6 Bilder und Bitmaps

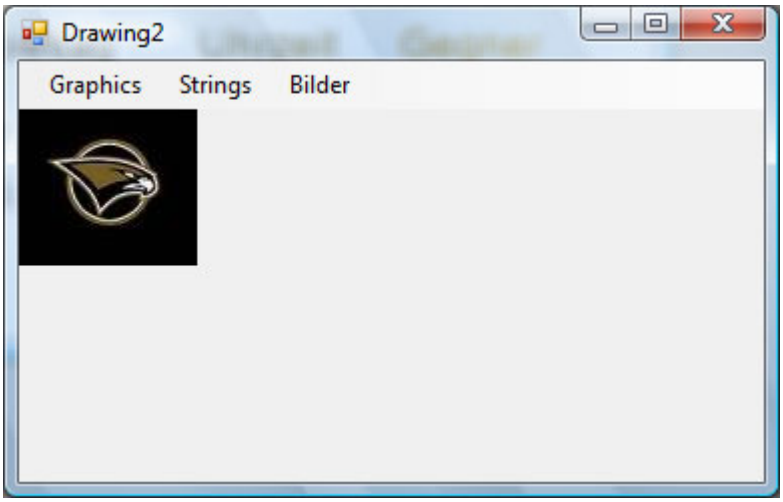
Auch Bilder lassen sich in GDI+ mit wenigen Zeilen Code erzeugen. Unterstützt werden dabei folgende Bildformate: Bitmap (BMP), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), Exchangeable Image Files (Exif), Portable Network Graphics (PNG), Icons (ICO) und Tag Image File Format (TIFF). Weiterhin unterstützt GDI+ mit der Metafile-Class auch Vektorgrafiken vom Dateityp Windows Metafile Format (WMF), Enhanced Metafile (EMF) und EMF+.

Aber wie bringt man nun ein Bild auf die Oberfläche, ohne eine `PictureBox` auf die Form zu ziehen und deren Eigenschaften festzulegen? Natürlich – wie sollte es auch anders sein in der GDI+-Welt – brauchen wir wieder ein `Graphics`-Objekt. Und da wir ein Bild zeichnen wollen, rufen wir die Methode `DrawImage()` auf und übergeben dabei ein `Image`-Objekt. Dieses erhalten wir, indem wir die statische Methode `FromImage()` der abstrakten Basisklasse `System.Drawing.Image` aufrufen. Die Methode gibt uns dann ein Objekt vom Typ `System.Drawing.Image` zurück, nachdem wir den Speicherort eines vorhandenen Bilds angegeben haben.

```
g.DrawImage(Image.FromFile _
  (Application.StartupPath & "\logo.jpg"), 0, 20)
```

Dies funktioniert für alle oben angegebenen Dateiformate identisch. Dabei erhalten Sie die in Abbildung 12.24 gezeigte Ausgabe.

Abbildung 12.24
Ausgabe eines Bilds



Nachdem Sie jetzt wissen, wie man Bilder per Programmcode öffnet, möchten Sie jetzt bestimmt einen Schritt weitergehen und das Bild manipulieren. Nehmen wir das Bild von vorhin. Wie müssten wir vorgehen, falls wir nur den Kopf des Habichts in dem Bild anzeigen wollen? Nun, man nehme ein `Bitmap`-Objekt, rufe die `Clone()`-Methode auf und übergebe ein Rechteck, das die Größe des Kopfs umfasst, und positioniere es mit der entsprechenden Höhe und Breite auf Kopfhöhe des Logos. Anschließend übergibt man den Bildausschnitt als Bitmap der `DrawImage()`-Methode und schon ist das Problem gelöst, wie in Listing 12.32 dargestellt.

Falls Sie diesen Ausschnitt jetzt noch vergrößern oder verkleinern wollen, so übergeben Sie der `DrawImage()`-Methode statt der Koordinaten einfach ein Rechteck mit der gewünschten Größe. Dabei muss man jedoch aufpassen, dass das Seitenverhältnis der Bildbreite und der Bildhöhe angepasst ist, es sei denn, eine Verzerrung des Ausgangsfotos ist gewünscht. Dies wird im unteren Abschnitt des Listing 12.32 durchgeführt.

Listing 12.32
Bildausschnitt zeichnen

```
Dim g As Graphics = e.Graphics
Dim rect As New Rectangle(40, 20, 30, 30)
Dim bmp As Bitmap = New Bitmap(Application.StartupPath & _
  "\logo.jpg").Clone(rect, Imaging.PixelFormat.DontCare)
g.DrawImage(bmp, 0, 20)
Dim r As New Rectangle(50, 50, 100, 100)
g.DrawImage(bmp, r)
```

Nachdem Sie das Bild nun erstellt beziehungsweise skaliert oder anderweitig verändert haben, wollen Sie dieses natürlich auch noch abspeichern. Doch wie speichern wir jetzt eigentlich das Bild, das sich ja bisher nur im Arbeitsspeicher befindet? Nun, dafür hat die `Bitmap`-Klasse auch eine Lösung parat. Übergeben Sie dieser doch einfach das Bild, sagen Sie ihr noch, wo sie es denn mit welchem Namen, mit welcher Endung und welchem Format speichern soll, und rufen Sie die Methode `Save()` auf:

```
bmp.Save("C:\kopf.gif", ImageFormat.Gif)
```

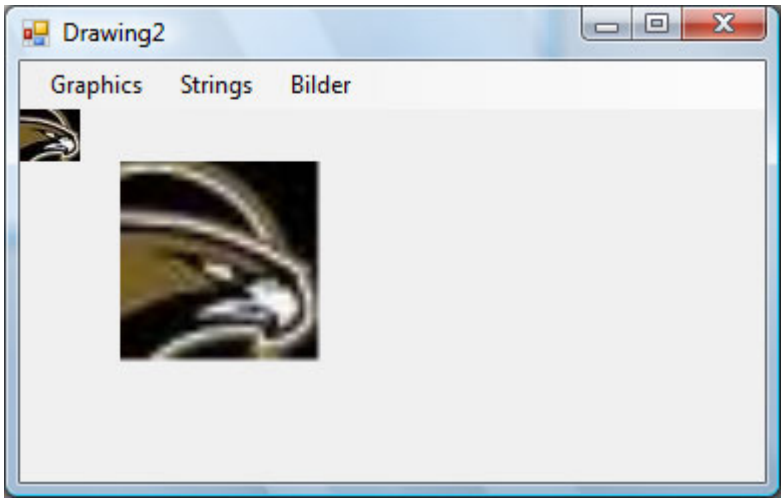


Abbildung 12.25
Gezeichneter Bildausschnitt

Dabei stellt Ihnen die `ImageFormat`-Klasse alle bereits aufgelisteten Datentypen als Eigenschaften zur Verfügung.

12.4 Serialisierung

Unter **Serialisierung** versteht man das Umwandeln einer komplexen Objektstruktur in einen einfachen Bytestrom. Man spricht dabei auch manchmal davon, das Objektgebilde in einen Bytestrom »flach zu klopfen«. Dies wird dann benötigt, wenn man zum Beispiel den Zustand eines Objekts speichern oder über ein Netzwerk für eine andere Anwendung übertragen will. Auch das Erzeugen einer Kopie eines Objekts, das Klonen von Objekten, ist mittels Serialisierung möglich.

Info

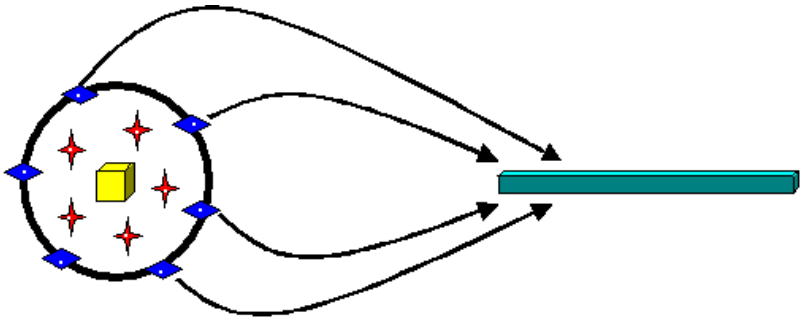
Bei der Serialisierung werden nur die Daten des Objekts in den Bytestrom übertragen. Funktionalität nicht! Es wird nur der Zustand des Objekts serialisiert.

Der Vorgang, um aus diesem flachen Bytestrom wieder ein Objekt zu erstellen, heißt **Deserialisierung**.

Dabei gibt es unterschiedliche Arten der Serialisierung.

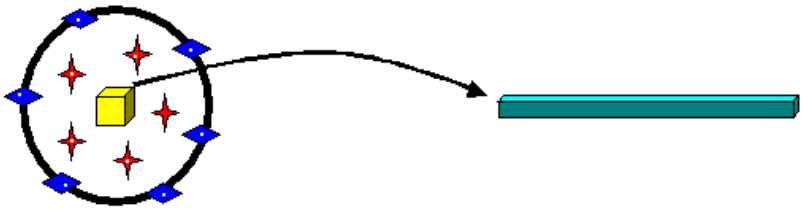
Shallow-Serialisierung, die oberflächliche Serialisierung. Das bedeutet, es werden nur die öffentlich zugänglichen Daten eines Objekts serialisiert, also nur die Eigenschaften, die einen öffentlichen Lese- und Schreibzugriff besitzen. Diesen Prozess nennt man deshalb nach dem englischen Wort shallow (zu Deutsch: oberflächlich). Er ist in Abbildung 12.26 dargestellt.

Abbildung 12.26
Shallow-Serialisierung



Deep-Serialisierung, die tiefe Serialisierung. Unter *Deep Serialization* versteht man das Transformieren aller Daten, auch der privaten und geschützten Eigenschaften des Objekts. Es wird somit eine exakte Kopie des Objekts erzeugt. Dies soll Abbildung 12.27 veranschaulichen.

Abbildung 12.27
Deep Serialization



Des Weiteren kann man nicht nur ein Objekt serialisieren, sondern einen gesamten Objektgraphen. Es gibt sehr viele Objekte, die als Eigenschaft einen Verweis auf ein anderes Objekt besitzen. Wenn das Objekt jetzt serialisiert wird, dann macht es relativ wenig Sinn, die Adresse des referenzierten Objekts mit zu serialisieren, da diese Objektadresse in einem anderen Kontext völlig sinnlos ist. Viel besser wäre es, wenn der gesamte Objektgraph serialisiert wird. Das bedeutet, dass alle abhängigen Objekte ebenso serialisiert werden. Diesen Vorgang nennt man die Serialisierung eines Objektgraphen, wie Sie in Abbildung 12.28 sehen.

Für das Format, in das die Daten in den Bytestrom geschrieben werden, gibt es innerhalb des .NET Framework drei verschiedene Formatierer.

- XMLSerializer

  Befindet sich im Namespace System.Xml.Serialization und erstellt den Bytestrom im XML-Format. Der XMLSerializer kann nur öffentliche Daten serialisieren (Shallow Serialization).

- BinaryFormatter

  Befindet sich im Namespace System.Runtime.Serialization.Formatters.Binary und erstellt einen binären Bytestrom. Der BinaryFormatter erzeugt Daten in einem sehr kompakten Format und unterstützt Deep Serialization eines Objektgraphen.

- SOAPFormatter

  Befindet sich im Namespace System.Runtime.Serialization.Formatters.Soap und erstellt einen Bytestrom im XML-Format innerhalb eines SOAP-Bodys. Der SOAPFormatter erzeugt Daten in lesbarem XML-Format und unterstützt Deep Serialization eines Objektgraphen. Um diesen Formatter zu nutzen, muss zu dem Projekt ein Verweis auf die Bibliothek *system.runtime.serialization.formatters.soap.dll* gesetzt werden. Der SOAPFormatter kann keine generischen Typen serialisieren.

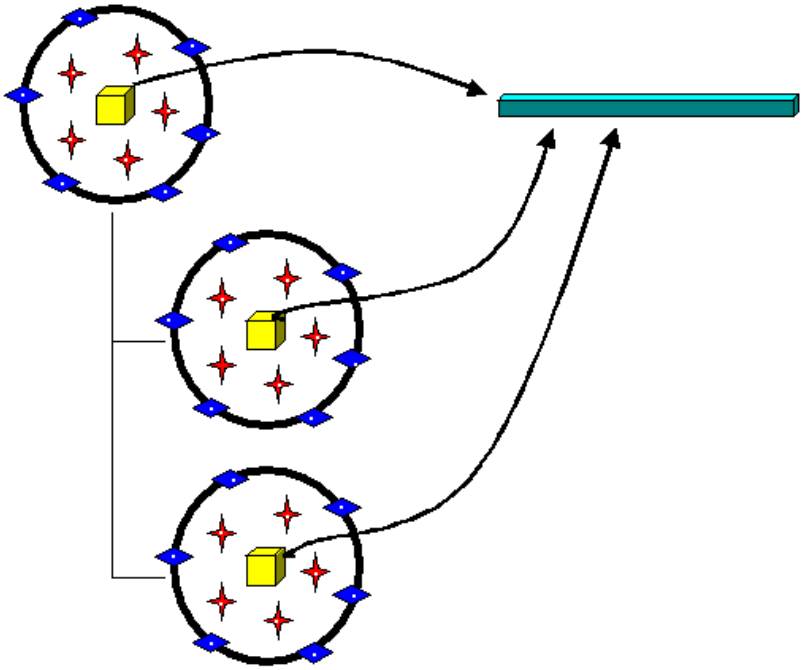


Abbildung 12.28
Serialisierung eines Objektgraphen

SOAP ist ein standardisiertes XML-basierendes Übertragungsprotokoll für strukturierte Informationen. Eine SOAP-Nachricht besteht aus einem SOAP-Envelope. Innerhalb dieses Envelope befinden sich ein optionaler SOAP-Header und ein SOAP-Body, in dem die tatsächlichen Informationen liegen.

Info

Um ein Objekt zu serialisieren, muss dieses Objekt auch als serialisierbar gekennzeichnet sein. Zu diesem Zweck gibt es im .NET Framework das `Serializable`-Attribut. Jedes Objekt, das serialisiert werden soll, muss dieses Attribut besitzen, ansonsten wird ein Laufzeitfehler ausgelöst.

Bei der Serialisierung eines Objektgraphen müssen auch alle abhängigen Objekte mit dem Attribut `Serializable` versehen sein.

Achtung

Außerdem kann man noch zwischen einer einfachen und einer benutzerdefinierten Serialisierung unterscheiden.

Bei der einfachen Serialisierung wird das Objekt automatisch vom .NET Framework serialisiert. Mit dem `NonSerialized`-Attribut können Sie bestimmte Felder aus der Serialisierung ausschließen. Wenn Objekte unterschiedliche Versionen haben, sich also die Objektdefinition geändert hat, werden Sie mit einfacher Serialisierung Probleme bekommen.

Diese Probleme können Sie mit einer benutzerdefinierten Serialisierung umgehen. Dabei können Sie genau angeben, wie die Serialisierung ausgeführt werden soll. Sie müssen zusätzlich auch noch das Interface `ISerializable` implementieren. Wenn Sie dieses Objekt später auch wieder benutzerdefiniert deserialisieren wollen, müssen Sie für diesen Zweck einen eigenen Konstruktor in der zu serialisierenden Klasse anlegen.

Im folgenden Beispiel definiere ich eine Klasse mit zwei öffentlichen Eigenschaften (eine davon schreibgeschützt) und einer privaten Variablen, die jedoch über eine Methode nach außen bekannt gemacht werden kann. Den Code für diese Klasse finden Sie in Listing 12.33.

Listing 12.33
Zu serialisierende Klasse

```
Public Class Konto

  Private mKontoNummer As Integer
  Private mInhaber As String
  Private mErstelltAm As Date

  Public Sub New(ByVal Inhaber As String)
    mErstelltAm = System.DateTime.Now
    Dim r As New Random
    mKontoNummer = _r.Next(100000000, 999999999)
    Me.Inhaber = Inhaber
  End Sub

  Public Property Inhaber() As String
    Get
      Return mInhaber
    End Get
    Set(ByVal value As String)
      mInhaber = value
    End Set
  End Property

  Public ReadOnly Property KontoNummer() As Integer
    Get
      Return mKontoNummer
    End Get
  End Property

  Public Function WannErstellt() As Date
    Return mErstelltAm
  End Function
End Class
```

Ich will innerhalb der Klasse zwei zusätzliche Methoden anbieten, um zum einen das Objekt mittels SOAP in eine Datei zu serialisieren und zum anderen mit dem binären Formatierer eine Kopie des Objekts zu erzeugen. Dazu muss ich zuerst noch einen Verweis auf die Bibliothek *System.Runtime.Serialization.Formatters.Soap* setzen.

Ich importiere dann die beiden Namespaces für die Formatierer und versehe die Klasse mit dem `Serializable`-Attribut. Außerdem importiere ich noch den Namespace `System.IO`, um Streams zu haben, in die der Bytestrom geschrieben wird.

Die SOAP-Serialisierung führe ich in der Methode `Save()` durch, an die ein Dateiname übergeben wird, in dem das serialisierte Objekt mittels eines `FileStream`-Objekts gespeichert werden soll. In einer Methode `Clone()` erzeuge ich eine Kopie des Objekts mit dem kompakten binären Formatierer und einem `MemoryStream`. Der Inhalt des MemoryStream wird anschließend deserialisiert und als Kopie der ursprünglichen Klasse zurückgegeben.

Listing 12.34
Serialisierung

```
Imports System.Runtime.Serialization.Formatters.Soap
Imports System.Runtime.Serialization.Formatters.Binary
Imports System.IO
<Serializable()> Public Class Konto

  Public Function Clone() As Konto
    Dim bf As New BinaryFormatter
    Dim ms As New MemoryStream
    bf.Serialize(ms, Me)
   'sonst wird am Ende des MemoryStreams begonnen
    ms.Position = 0
    Dim newKonto = CType(bf.Deserialize(ms), Konto)
    Return newKonto
  End Function

  Public Sub Save(ByVal dateiName As String)
    Dim sf As New SoapFormatter
    Dim fs As New FileStream(dateiName, FileMode.Create)
    sf.Serialize(fs, Me)
    fs.Close()
  End Sub
…
End Class
```

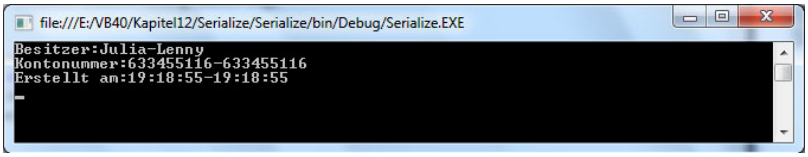
In einer Konsolenanwendung rufen wir beide Methoden auf, um uns davon zu überzeugen, dass auch alles funktioniert. Den Programmcode der `Sub Main()` sehen Sie in Listing 12.35.

Listing 12.35
Aufruf der Serialisierungsmethoden

```
Dim k As New Konto("Lenny")
k.Save("C:\temp\konto.xml")
Dim k2 As Konto = k.Clone
'als Beweis für unterschiedliche Instanzen
k2.Inhaber = "Julia"
'Ausgaben sind unterschiedlich
Console.WriteLine("Besitzer: " & k2.Inhaber & "-" & k.Inhaber)
'sind gleich
Console.WriteLine("Kontonummer: " & _
    k2.KontoNummer & "-" & k.KontoNummer)
Console.WriteLine("Erstellt am: " & _
    k2.WannErstellt.ToLongTimeString & "-" & _
    k.WannErstellt.ToLongTimeString)
```

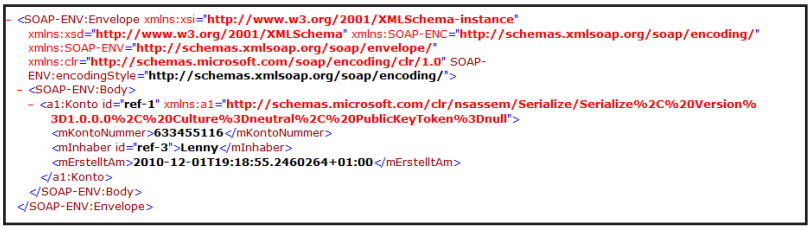
Die zugehörige Bildschirmausgabe sehen Sie in Abbildung 12.29.

Abbildung 12.29
Bildschirmausgabe des Serialisierungsbeispiels



Die erzeugte Datei *Konto.xml* sehen Sie in Abbildung 12.30.

Abbildung 12.30
Konto.xml serialisiert durch den SoapFormatter



Wie Sie sehen, haben wir bei beiden Formatierungen jeweils alle Daten, auch die privaten, mit serialisiert.

Zwecks der Vollständigkeit sehen Sie im abschließenden Listing 12.36 noch den Einsatz des `XmlSerializer`.

Listing 12.36
XmlSerializer im Einsatz

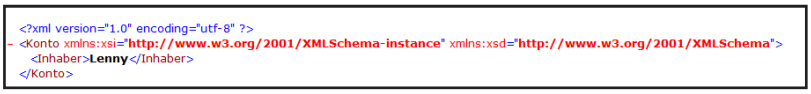
```
Dim k As New Konto("Lenny")
Dim serializer As New XmlSerializer(GetType(Konto))
Dim writer As Xml.XmlWriter = _
    Xml.XmlWriter.Create("C:\temp\Konto2.xml")
serializer.Serialize(writer, k)
writer.Close()
```

Achtung

Damit der `XmlSerializer` problemlos funktioniert, benötigen Sie noch einen paramterlosen Konstruktor in der Klasse `Konto`.

Die erzeugte Datei *Konto2.xml* sehen Sie in Abbildung 12.31.

Abbildung 12.31
Konto2.xml serialisiert mit dem XmlSerializer



Wie Sie in Abbildung 12.31 erkennen können, wird nur noch die Eigenschaft *Inhaber* serialisiert. Der SoapFormatter serialisierte zusätzlich die Eigenschaften *mKontonummer* und *mErstelltAm*. *Kontonummer* ist eine schreibgeschützte Eigenschaft und *mErstelltAm* ist eine private Variable, beides Dinge, die der *XmlSerializer* nicht serialisieren kann.

12.5 Multithreading

Neben der Vererbung war **Multithreading** eine der wichtigsten Neuerungen bei der Einführung von Visual Basic .NET. In Visual Basic 6 war es nur möglich, Programme in einem einzigen **Thread** ablaufen zu lassen, doch mittlerweile gibt es die Möglichkeit, eine Anwendung in vielen verschiedenen `Threads` ablaufen zu lassen. Ein Thread – zu Deutsch Faden – ist ein eigenständiger Programmpfad, sozusagen ein Programm im Programm.

Mit der aktuellen Version 4.0 des .NET Framework gab es nochmals erhebliche Verbesserungen und Vereinfachungen bei der Programmierung von Multithreading-Anwendungen.

Die Programmierung von mehreren `Threads` ermöglicht es Ihnen, in Ihrer Anwendung verschiedene Programmabläufe so aufzuteilen, dass diese größtenteils unbeeinträchtigt voneinander erledigt werden können. Somit kann die Ausführungszeit von Programmen wesentlich verbessert werden, da bestimmte Prozesse (wie eigene Unterprogramme) in eigenen Threads ablaufen können und der Hauptthread (das Hauptprogramm) nicht auf die Beendigung dieses Prozesses warten muss. Für die Verwaltung der Threads und die Ausführung und Zuweisung von Prozessorzeit an die Threads ist das Betriebssystem zuständig.

Achtung

Alle Threads einer Applikation teilen sich denselben Adressraum; sie greifen somit auf dieselben Objekte und Variablen zu.

Das .NET Framework stellt Ihnen einen `Threading`-Namespace zur Verfügung, mit dem Sie `Threads` starten und notwendige Synchronisationen durchführen können.

Tipp

Beachten Sie aber, dass Multithreading sehr wohl überlegt sein will, denn bei ungeschicktem Einsatz des Threadings kann Ihr Programm auch langsamer werden, da sich verschiedene Threads blockieren könnten. Außerdem tauchen neue Probleme auf, die vor allem für nicht allzu erfahrene Programmierer kaum zu lösen sind, da sie extrem schwer zu debuggen sind.

Um nun verschiedene Threads in Ihr Programm einzubauen, brauchen Sie das `Thread`-Objekt aus dem Namespace `System.Threading`. Mit diesem Objekt können Sie verschiedene `Threads` starten und beenden. Um diese Klasse nutzen zu können, empfiehlt es sich, mit dem `Imports`-Befehl auf diesen `Namespace` zu verweisen. Somit müssen Sie nicht immer den voll qualifizierenden Klassennamen verwenden.

```
Imports System.Threading
```

Um ein `Thread`-Objekt zu erzeugen, benutzen Sie wie gewohnt den `New`-Operator:

```
Dim myThread As Thread = New Thread( _
  New ThreadStart (AddressOf Klasse.Methode))
```

Kürzer kann das Thread-Objekt auch folgendermaßen erzeugt werden:

```
Dim myThread As New Thread(AddressOf Klasse.Methode)
```

Mit dieser Anweisung haben Sie einen neuen Thread angelegt, der die Funktionalität in der Methode der angegebenen Klasse (Klasse.Methode()) beim Aufruf des Threads startet. Die Adresse der Methode wird durch den AddressOf-Operator an den Konstruktor der Thread-Klasse übergeben. Mit der Methode Start() können Sie dann den Thread starten:

```
MyThread.Start()
```

Im folgenden Beispiel sehen Sie, wie man mit Multithreading innerhalb einer Anwendung zwei anstehende Arbeiten mit der Benutzung von zwei Threads gleichzeitig ermöglichen kann. In einem Konto sind dabei zwei Methoden EinzahlungenBuchen() und AuszahlungenBuchen() implementiert. Dabei werden jeweils zehn Millionen Buchungen parallel durchgeführt. Da alle Buchungen mit demselben Betrag durchgeführt werden, sollte der Kontostand, die einzige Eigenschaft der Klasse, nach den Buchungen wieder auf dem ursprünglichen Stand sein. Sie sehen den Programmcode in Listing 12.37.

Listing 12.37 Kontodefinition

```
Public Class Konto
  Protected mKontostand As Double
  Public Property Kontostand() As Double
    Get
      Return mKontostand
    End Get
    Set(ByVal Value As Double)
      mKontostand = Value
    End Set
  End Property

  Public Sub EinzahlungenBuchen()
    For i As Integer = 1 To 10000000
      mKontostand = mKontostand + 2
    Next i
    Console.WriteLine _
      ("Der neue Kontostand nach den Einzahlungen lautet: " & _
        mKontostand.ToString)
  End Function

  Public Sub AuszahlungenBuchen()
    For i As Integer = 1 To 10000000
      mKontostand = mKontostand - 2
    Next i
    Console.WriteLine _
      ("Der neue Kontostand nach den Auszahlungen lautet: " & _
        mKontostand.ToString)
  End Sub
End Class
```

In einem Hauptprogramm innerhalb einer Konsolenanwendung rufen wir jetzt die Buchungen auf und schauen uns an, wie die beiden Threads parallel ausgeführt werden. Die Sub Main() ist in Listing 12.38 dargestellt.

```
Dim Konto1 As Konto = New Konto()
Konto1.Kontostand = 8000
Dim myThread As Thread = New Thread _
    (AddressOf Konto1.EinzahlungenBuchen)
Dim myThread2 As Thread = New Thread _
    (AddressOf Konto1.AuszahlungenBuchen)
myThread.Start()
myThread2.Start()
myThread.Join()
myThread2.Join()
Console.WriteLine("Kontostand nach den Buchungen: " & _
  Konto1.Kontostand.ToString("c"))
Console.ReadLine()
```

Listing 12.38
Aufruf der Threads

In der `Main()`-Methode wird ein `Konto`-Objekt erstellt und der Kontostand über die entsprechende Eigenschaft gesetzt. Danach werden zwei Threads erstellt, die dann auch gestartet werden. Beide Threads laufen parallel und während im ersten Thread Einzahlungen gebucht werden, bucht der zweite Thread bereits die Auszahlungen. Nachdem die beiden Threads mit `Start()` gestartet wurden, wird im Hauptprogramm mit der Methode `Join()` gewartet, bis die Threads fertig abgearbeitet wurden.

Abbildung 12.32 zeigt die zugehörige Bildschirmausgabe. Wie wir wahrscheinlich nicht anders erwartet haben, ist nach den Buchungen der Kontostand genauso, wie er ursprünglich war. Obwohl wir zuerst die Einzahlungen gebucht haben, wird zuerst die Ausgabe der Auszahlungen am Bildschirm angezeigt. Das kann jedoch bei mehreren Tests unterschiedlich sein.

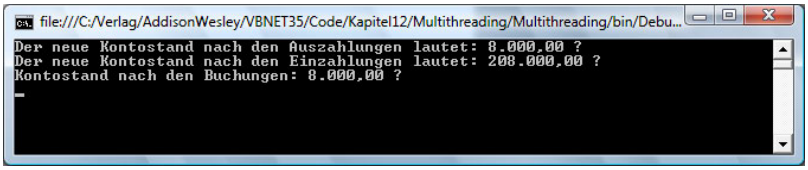


Abbildung 12.32
Ausgabe des Multithreading-Programms mit kleinem Schleifenzähler

Achtung

Mir ist bewusst, dass die Aufgabenstellung in diesem Beispiel nicht unbedingt exemplarisch für den Einsatz von Multithreading ist, jedoch lassen sich meines Erachtens die Implementierung von Multithreading und die damit zusammenhängenden Probleme gerade an diesem einfachen Beispiel sehr schön darstellen.

Der Thread könnte auch mit der Methode `Abort()` angehalten werden. Mit der statischen Methode `Sleep()` kann er ebenfalls für eine bestimmte Zeit angehalten werden. Vor dem Starten eines Threads lassen sich noch zwei wichtige Eigenschaften des `Thread`-Objekts einstellen:

- `IsBackground` gibt an, ob der Thread im Hintergrund laufen soll. Hintergrundthreads werden mit dem Hauptthread beendet.
- `Priority` gibt an, in welcher Priorität der `Thread` laufen soll (es empfiehlt sich, eine niedrigere Priorität zu wählen als den Hauptthread).

Beispiel:

```
mThread.IsBackground = True
mThread.Priority = Threading.ThreadPriority.BelowNormal
```

So, und nun wollen wir betrachten, was passiert, wenn wir die Schleife innerhalb der Einzahlen- und Auszahlenfunktionalität auf Hundertmillionen erhöhen.

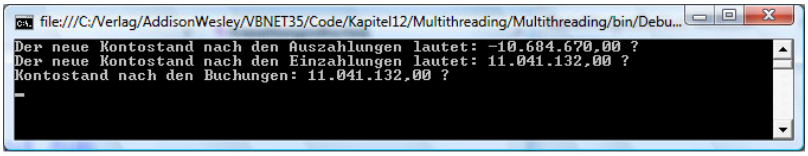


Abbildung 12.33 Ausgabe des Multithreading-Programms mit einem hohen Schleifenzähler

Wie Sie sehen, stimmt unser Ergebnis nicht mehr und – was noch schlimmer ist – bei jedem Aufruf des Programms sind die angezeigten Werte unterschiedlich. Und genau das ist das Problem von Multithreading, denn beide Threads greifen gleichzeitig auf dieselbe Variable `mKontostand` zu.

12.5.1 Thread-Modelle

Unter .NET gibt es drei verschiedene Threading-Modelle:

- Single Threaded
- Apartment Model Threading (STA)
- Free Threaded (MTA Multithreaded Apartment)

In Visual Basic 6 konnten Sie die Modelle *SingleThreaded* und *Apartment Model Threaded* nutzen. Beide Konzepte eliminieren Multithreading-Probleme, wie wir sie gerade hatten, dadurch, dass sie sämtliche globalen Variablen (die von den verschiedenen Threads gemeinsam genutzt werden können) auf lokale Variablen innerhalb der Threads kopieren. Unter .NET haben wir aber ein *FreeThreaded*-Multithreading und müssen uns um sämtliche Synchronisationsprobleme und Sicherheitsmechanismen selbst kümmern.

Beim obigen Beispiel addieren beziehungsweise subtrahieren wir auf die Variable *mKontostand* jeweils den Wert 2. Multithreading bedeutet, dass der Prozessor die verschiedenen Threads nach bestimmten Verfahren abarbeitet. Das Wechseln der Threads erfolgt so schnell, dass wir es als paralleles Abarbeiten der verschiedenen Programme empfinden. Der Prozessor arbeitet jedoch den für ihn übersetzten IL-Code ab, also reinen Maschinencode. Unsere Anweisung `mKontostand = mKontostand + 2` besteht jedoch aus mehreren Maschinencode-Anweisungen.

Eine einfache Übersetzung in Assembler würde bereits Folgendes aus unserer Zeile machen:

- Lade den Wert der Variablen `mKontostand` in das Register.
- Addiere den Wert 2 auf das Register.
- Speichere den Wert des Registers in der Variablen `mKontostand`.

Das Problem an der Sache ist nun der Wechsel des Threads zu einem beliebigen Zeitpunkt. Das bedeutet, dass nach dem Laden der Variablen `mKontostand` in das Register ein Wechsel des Threads erfolgen kann.

Thread2 führt seine Aufgabe durch und speichert in der Variablen den berechneten Wert. Nun kommt wieder Thread1 an die Reihe und führt seine Anweisungen fort. Da Schritt 1 bereits abgearbeitet wurde, wird mit Schritt 2 weitergemacht. Auf das Register wird jetzt 2 aufaddiert und zurück in die Variable geschrieben. Sämtliche Auszahlungen werden in diesem Fall überschrieben. Und genau für diese Art von Problemen sollten Sie ein Verständnis besitzen, bevor Sie mit Multithreading so richtig beginnen.

12.5.2 Thread-Sicherheit

Wir müssen jetzt versuchen, genau dieses Verhalten nicht zuzulassen. Das können wir dadurch tun, indem wir die Anweisung synchronisieren. Wir setzen mit **Synchronisation** ein Lock auf die Variablen innerhalb des synchronisierten Blocks. Somit ist gewährleistet, dass kein anderer Thread den kritischen Block ausführen kann, solange dieser Block nicht abgeschlossen wurde, was im Endeffekt bedeutet, solange der Wert der Variablen nicht wieder zurückgeschrieben wurde.

Der Befehl hierfür ist die `SyncLock`-Anweisung, die als Parameter eine Referenzvariable benötigt.

Diese Anweisung gibt uns nun einerseits die Sicherheit, dass die weiter oben beschriebenen Fehler nicht mehr auftreten können. Jedoch verlangsamt sie die Laufzeit der Anwendung, da sich jetzt Threads blockieren können und das erneute Starten eines blockierten Threads eine zeitaufwändige Sache ist. Die Synchronisation in unseren beiden Methoden sehen Sie in Listing 12.39.

Listing 12.39
Synchronisationsbeispiel

```
Public Sub EinzahlungenBuchen()
  For i As Integer = 1 To 100000000
    SyncLock Me
      mKontostand = mKontostand + 2
    End Synclock
  Next i
  Console.WriteLine _
    ("Der neue Kontostand nach den " & _
     "Einzahlungen lautet: " & _
      mKontostand.ToString)
End Sub

Public Sub AuszahlungenBuchen()
  For i As Integer = 1 To 100000000
    SyncLock Me
      mKontostand = mKontostand - 2
    End Synclock
  Next i
  Console.WriteLine _
    ("Der neue Kontostand nach den " & _
     "Auszahlungen lautet: " & _
      mKontostand.ToString)
End Sub
```

Wenn Sie das Programm jetzt starten, werden Sie feststellen, dass wieder das korrekte Ergebnis ermittelt wird, die Ausführung jedoch wesentlich länger dauert. Das liegt daran, dass sich die Threads jetzt gegenseitig blockieren und nicht ausgeführt werden können, wenn sie vom Betriebssystem Prozessorzeit zugewiesen bekommen.

Achten Sie darauf, dass Sie keine Deadlock-Situationen erzeugen. Unter einem **Deadlock** versteht man, dass zwei verschiedene Threads sich gegenseitig blockieren. Thread1 will ProzedurA abschließen, muss dazu aber ProzedurB aufrufen. ProzedurB ist aber momentan gesperrt, weil Thread2 ProzedurB noch nicht abgeschlossen hat, da der Thread gerade beim Versuch, auf ProzedurA zuzugreifen, blockiert.

Denken Sie dabei immer an Murphys Gesetz: »Wenn ein Deadlock auftreten könnte, wird er dies auch tun!«

Der Ausdruck *Threadsicherheit* bedeutet, dass ein Objekt auch dann in einem gültigen Zustand bleibt, wenn mehrere Threads darauf zugreifen.

Als Alternative zur oben genannten Syntax steht Ihnen auch noch die mächtigere Klasse `Monitor` aus dem Namespace `System.Threading` zur Verfügung.

Aus dieser Klasse werden zwei statische Methoden benutzt, `Enter()` und `Exit()`. `Enter()` sperrt den folgenden Codeblock für weitere Zugriffe. Falls ein anderer Thread das Objekt schon gesperrt hat, blockiert `Enter()` den Codeblock so lange, bis er vom vorherigen Thread wieder freigegeben wird. Mit der Methode `Exit()` lässt sich die Sperre wieder aufheben.

Anstatt des Konstrukts `SyncLock … EndSyncLock` würden Sie mit der Klasse `Monitor` den zu synchronisierenden Block zwischen die Anweisungen `Monitor.Enter() … Monitor.Exit()` schreiben; wobei hier die Methoden `Enter()` und `Exit()` auch wieder als Parameter ein Referenzobjekt benötigen.

Eine andere Möglichkeit, Anwendungen threadsicher zu machen, bietet die Klasse `Mutex`. Der Ausdruck `Mutex` wurde aus den Wörtern `mutually exclusive` gebildet. Die `Mutex`-Klasse ist ebenfalls im Namespace `System.Threading` definiert. Ein `Mutex` kann immer nur zu einem Thread gehören. Für ein `Mutex`-Objekt stehen drei Konstruktoren zur Verfügung:

- Mutex()
- Mutex(boolean)
- Mutex(boolean, String)

Der erste Konstruktor erstellt einen neuen `Mutex` und macht den aktuellen Thread zu seinem Besitzer. Dies hat zur Folge, dass der Mutex auch gleich vom aktuellen Thread gesperrt wird.

Der zweite Konstruktor initialisiert einen neuen Mutex mit einem booleschen Wert, der angibt, ob der aufrufende Thread auch Besitzer des Mutex sein soll.

Der dritte Konstruktor initialisiert einen neuen Mutex mit einem booleschen Wert, der angibt, ob der aufrufende Thread auch von vornherein Besitzer des Mutex sein soll, sowie einen String, der dem Mutex einen Namen gibt.

Diese Problematiken und deren Lösungen sollten Ihnen sehr wohl bewusst sein, wenn Sie Multithreading in Ihrer Anwendung einbauen wollen. Das hier gezeigte Beispiel zeigt sehr schön die Probleme, die beim Multithreading auftreten können. Wie Sie bemerkt haben, ist dadurch die Anwendung wesentlich langsamer geworden als ohne Multithreading. Sie hätten also genau das Gegenteil dessen bewirkt, was Sie eigentlich erreichen wollten.

Tipp

Vermeiden Sie bei prozessorintensiven Berechnungen den Einsatz von Multithreading, es sei denn, Sie setzen bei der Entwicklung Multiprozessormaschinen voraus.

Nichtsdestotrotz gibt es auch sehr schöne Beispiele, wann Multithreading lohnenswert ist. Dazu gehört das Drucken im Hintergrund oder wenn man Daten aus verschiedenen Datenquellen abruft und dies parallel durchführen kann.

Auch wenn Multithreading – zumindest das Starten und Initialisieren von Threads – im .NET Framework sehr einfach implementiert ist, sollten Sie sich jedes Mal reiflich überlegen, ob der Anwendungsfall, den Sie gerade im Kopf haben, wirklich für Multithreading geeignet ist.

12.5.3 Parallelisierung im .NET Framework 4.0

Durch die immer weitergehende Entwicklung der Hardwarehersteller im Bereich der Entwicklung von Mehrkernprozessoren können viele Aufgaben parallel und somit schneller und effizienter abgearbeitet werden. Doch die auszuführenden Programme müssen natürlich für diese parallele Ausführung geschrieben und designt werden. Im Gegensatz zu Multithreading werden bei Parallelprogrammierung Programmteile wirklich parallel abgearbeitet. Bei Multithreading wechselt der zu bearbeitende Prozess so schnell, dass es nur so aussieht, als würde die Aufgabe wirklich parallel abgearbeitet.

Das .NET Framework 4.0 zielt genau auf dieses Design ab und führt zahlreiche Verbesserungen und Erweiterungen für diese Art der parallelen Entwicklung ein.

Seien Sie jedoch gewarnt, auch durch diese parallelen Erweiterungen im Framework entsteht eine höhere Komplexität in der Entwicklung und die gerade beschriebenen Probleme im Bereich der Synchronisierung werden sich nicht in Luft auflösen.

Parallele Schleifen

Im neuen Namespace *System.Threading.Tasks* gibt es ein *Parallel*-Objekt, mit dem Schleifen parallel ausgeführt werden können.

Mit diesem Objekt können sowohl eine *For*- wie auch eine *For Each*-Schleife parallel ausgeführt werden.

Betrachten Sie bitte den Code in Listing 12.40.

Listing 12.40
Eine parallele For-Schleife

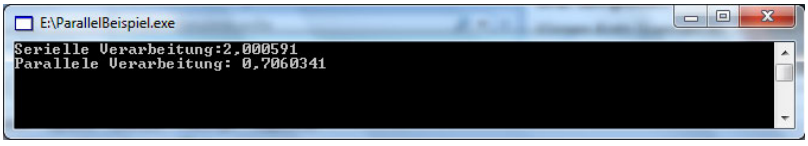
```
Imports System.Threading.Tasks
Module Module1
  Sub Main()
    Dim watch As New Stopwatch
    watch.Start()

    'serielle Ausführung
    For i = 1 To 100
        GetZufallszahl(i)
    Next
    Console.WriteLine("Serielle Verarbeitung: " &
                      watch.Elapsed.TotalSeconds.ToString())
    watch.Restart()
    Parallel.For(1, 101, AddressOf GetZufallszahl)
    Console.WriteLine("Parallele Verarbeitung: " &
                      watch.Elapsed.TotalSeconds.ToString())
     Console.ReadLine()
  End Sub
  Function GetZufallszahl(ByVal i As Integer) As Double
     Dim r As New Random(i)
     System.Threading.Thread.Sleep(20)
     Return r.NextDouble
  End Function
End Module
```

In diesem kleinen Beispiel wird zuerst in einer herkömmlichen Schleife 100 Mal eine Zufallszahl generiert. Die Generierung der Zufallszahl dauert durch die *Sleep*-Anweisung in der Methode *GetZufallszahl()* ca. 20 Millisekunden, also bei 100 Durchläufen in etwa zwei Sekunden.

Danach verwenden wir die neue *Parallel.For()*-Schleife. Wie der Bildschirmausdruck in Abbildung 12.34 beweisen wird, wird diese Schleife parallel ausgeführt, läuft also deutlich schneller als in zwei Sekunden ab.

Diese Überladung der *Parallel.For()*-Schleife erwartet dabei drei Parameter: den Startwert der Schleife, den Endwert der Schleife und die Adresse der Methode, die ausgeführt werden soll. Wie Sie sehen, ist der Endwert in diesem Beispiel 101, die Schleife wird also so lange durchlaufen, wie der Schleifenzähler kleiner als dieser Wert ist (angelehnt an die C#-Syntax einer *for*-Schleife).

Abbildung 12.34
Bildschirmausgabe der Parallel.For-Schleife



Das Ergebnis stimmt doch sehr positiv. Ohne großen Aufwand konnte man einen deutlichen Performancegewinn verbuchen. Doch leider muss ich die Euphorie gleich ein bisschen bremsen. Im folgenden Listing 12.41 habe ich eine Variable *summe* hinzugefügt, die bei jeder Berechnung der Zufallszahl um den Schleifenzähler addiert. Außerdem wird die Schleife jetzt 10.000 Mal

durchlaufen und der *Sleep* wurde etwas heruntergesetzt, damit die Wartezeit nicht so lange wird. Eigentlich müsste die Summe sowohl bei der seriellen Verarbeitung wie auch bei der parallelen Verarbeitung um 50005000 (laut Carl-Friedrich Gauss 10000*10001/2) erhöht werden, also am Ende den Wert 100010000 haben. Wie Sie in der Abbildung 12.35 sehen, tut es dies leider nicht, denn obwohl wir hier explizit keinen Thread erzeugt haben, treten dieselben Probleme wie beim Multithreading auf.

Listing 12.41
Synchronisationsproblem der Parallel.For-Schleife

```
Imports System.Threading.Tasks
Module Module1
    Private summe As Integer
    Sub Main()
       Dim watch As New Stopwatch
       watch.Start()

       'serielle Ausführung
       For i = 1 To 10000
          GetZufallszahl(i)
       Next
       Console.WriteLine(summe)
       Console.WriteLine("Serielle Verarbeitung: " &
                         watch.Elapsed.TotalSeconds.ToString())
       watch.Restart()
       Parallel.For(1, 10001, AddressOf GetZufallszahl)
       Console.WriteLine("Parallele Verarbeitung: " &
                         watch.Elapsed.TotalSeconds.ToString())
        Console.WriteLine(summe)
        Console.ReadLine()
    End Sub
    Function GetZufallszahl(ByVal i As Integer) As Double
        Dim r As New Random(i)
        summe += i
        System.Threading.Thread.Sleep(2)
        Return r.NextDouble
    End Function
End Module
```

Abbildung 12.35
Synchronisationsproblem

PLINQ – Parallel LINQ

Eine weitere Neuerung in diesem Zusammenhang ist die Einführung von Parallel LINQ (kurz PLINQ). Hierbei handelt es sich um die parallel verarbeitende Version von LINQ to Objects. Aktuell ist PLINQ nicht für LINQ to SQL und LINQ to Entities verfügbar, da hier die Arbeit ja auf der Datenbank vollzogen wird und somit nicht von .NET gesteuert werden kann.

Stellen Sie sich folgenden herkömmlichen LINQ-Ausdruck vor:

```
Dim laenderListe = From k In kundenListe
          Where k.Region = "BY" Select k.Region Distinct
```

Um diesen LINQ-Ausdruck parallel auszuführen, müssen Sie nur folgende Änderung durchführen:

```
Dim laenderListe = From k In kundenListe.AsParallel
          Where k.Region = "BY" Select k.Region Distinct
```

Nur durch das Anfügen der Extension ***AsParallel*** an die durchsuchende Liste wird diese Aufgabe parallel ausgeführt.

Beachten Sie bitte, dass Sie diese Erweiterung nur zum Lesen und Selektieren von Daten einsetzen. Also Finger weg beim Manipulieren der Daten!

12.6 Drucken und Reporting

Wenn Sie aus Ihrem Programm heraus Ausdrucke machen wollen, gibt es in Visual Basic 10 unterschiedliche Möglichkeiten, dies zu bewerkstelligen.

Sie können das zum einen über eine interne Klasse `PrintDocument` machen oder Sie benutzen integrierte Reportingtools wie *ReportViewer* oder *Crystal-Reports*.

Diese drei Möglichkeiten will ich Ihnen in diesem Abschnitt vorstellen. Natürlich gibt es noch viele weitere Reportingtools von Drittherstellern oder auch die Möglichkeit, mit Office-Automation Excel-Arbeitsmappen oder Word-Dokumente zu erstellen, aber das würde den Umfang dieses Buchs schlichtweg sprengen.

12.6.1 PrintDocument

»Wer in .NET zeichnen kann, der kann auch drucken.« Na ja, ganz so leicht ist es nicht, jedoch hat man schon fast gewonnen, wenn man mit dem `Graphics`-Objekt aus dem `System.Drawing`-Namespace umgehen kann. Denn das Drucken unterscheidet sich in der Methodik vom Zeichnen nur darin, dass wir einen anderen **Device-Kontext** haben, in unserem Fall den Drucker, statt wie üblich den Bildschirm. Die zentrale Klasse, die dabei sämtliche Informationen für den Drucker bereitstellt, heißt `PrintDocument` und befindet sich im Namensraum `System.Drawing.Printing`.

Um einen Druckvorgang zu starten, stellt uns die Klasse `PrintDocument` eine Methode namens `Print()` zur Verfügung, die ein Ereignis `PrintPage` auslöst, dessen Ereignishandler wir erst noch implementieren müssen.

Davor importieren wir aber, wie folgt, den passenden Namespace:

```
Imports System.Drawing.Printing
```

Zuerst verknüpfen wir das `PrintPage`-Event mit einem Ereignishandler innerhalb der `Form_Load()`-Methode.

```
Dim pd as New PrintDocument
AddHandler pd.PrintPage, AddressOf printme
pd.Print
```

und dann generieren wir einen Ereignishandler für unser `PrintPage`-Ereignis.

```
Private Sub printme(ByVal sender As Object, _
   ByVal e As PrintPageEventArgs)
   'hier die implementierung
End Sub
```

In der `Form_Load()`-Methode instanziieren wir ein neues `PrintDocument` und sagen diesem, dass beim Auslösen des `Print`-Ereignisses die Routine `printme` ausgeführt werden soll. Soweit würde noch gar nichts passieren, da wir noch keine Logik hinterlegt haben. Der Schlüssel zum Starten des Druckvorgangs steckt in dem `PrintPageEventArgs`, das dem Ereignishandler `printme` als Argument übergeben wird.

Tipp

Das Ganze hätten Sie übrigens auch lösen können, indem Sie ein `PrintDocument` von der Toolbox auf die Form ziehen und von diesem das Event `PrintPage` implementieren.

PrintPageEventArgs

Die `PrintPageEventArgs` beinhalten eine Eigenschaft `Graphics`, die das Grafik-Objekt für das Drucker-Device zurückgibt und mittels derer wir Ausgaben an den Drucker senden können, wie Sie in Listing 12.42 sehen:

Listing 12.42 Einfacher Befehl zum Ausdrucken

```
Private Sub printme(ByVal sender As Object, _
   ByVal e As PrintPageEventArgs)

   Dim rect As New Rectangle(10, 10, 200, 200)
   e.Graphics.DrawString("Hallo Welt", Me.Font, _
              Brushes.Black, rect)
   e.HasMorePages = False
End Sub
```

In diesem Fall würde »Hallo Welt« ausgedruckt werden, wobei das vorher definierte Rechteck die Fläche beschreibt, die dem String zur Verfügung steht. Diese Routine ist jedoch nur für das Drucken einer einzigen Seite zuständig, was bedeutet, dass Sie bei einem mehrseitigen Dokument selbst wissen müssen, nach wie vielen Zeichen beziehungsweise Sätzen eine neue Seite beginnt.

So, jetzt haben wir unseren ersten Ausdruck, doch wie können wir nun eine Druckvorschau oder etwa einen Dialog bezüglich der Seiten- und Druckereinstellungen implementieren, die uns bei anderen Programmen schon als Selbstverständlichkeit erscheinen? Dazu gibt es glücklicherweise ein paar weitere Klassen. Mit den entsprechenden Steuerelementen, die wir einfach aus der Toolbox ziehen, können wir dem Anwender seine gewohnten Dialoge zur Verfügung stellen.

PrintPreviewDialog

Das gewohnte Dialogfenster der Seitenansicht können wir aufrufen, indem wir eine Instanz der Klasse `PrintPreviewDialog` erzeugen, deren `Document`-Eigenschaft unser `PrintDocument` zuweisen und die Methode `ShowDialog()` aufrufen.

```
Dim PrintPreviewDialog1 As New PrintPreviewDialog
PrintPreviewDialog1.Document = pd
PrintPreviewDialog1.ShowDialog()
```

Voraussetzung dafür ist jedoch, dass das `PrintDocument`-Objekt `pd` innerhalb des gesamten Formulars bekannt ist.

PageSetupDialog

Auch für die Seiteneinstellungen gibt es eine Klasse im .NET Framework. Mit den Eigenschaften der `PageSetupDialog`-Klasse kann man sämtliche Felder des bekannten Dialogfensters Seite einrichten aktivieren beziehungsweise deaktivieren.

Listing 12.43 PageSetup-Einstellungen

```
Dim PageSetupDialog1 As New PageSetupDialog()
With PageSetupDialog1
  PageSetupDialog1.Document = pd
  PageSetupDialog1.AllowMargins = True
  PageSetupDialog1.AllowOrientation = False
  PageSetupDialog1.AllowPaper = False
  PageSetupDialog1.AllowPrinter = True
End With

If PageSetupDialog1.ShowDialog() = DialogResult.OK Then
  pd = PageSetupDialog1.Document
End If
```

In Listing 12.43 können Sie die Rändereinstellungen und die Druckerauswahl ändern, die anderen Einstellungen wie Papierauswahl und Hoch- oder Querformat dagegen erscheinen ausgegraut.

PrintDialog

Zum Absenden des Druckauftrags fehlt jetzt nur noch der Druckerdialog zur Auswahl des Druckers, des Druckbereiches und der Anzahl der Exemplare. Dazu verwenden wir eine Klasse mit dem Namen `PrintDialog`, dessen Eigenschaften eigentlich schon selbst beschreibend sind. Spielen Sie einfach ein wenig mit dieser Klasse, setzen Sie die vorhandenen Eigenschaften auf `True` oder `False` und schauen Sie, was passiert.

Drucken von mehreren Seiten

Angenommen, wir haben mehrere Seiten zusammenhängenden Text, die wir ausdrucken möchten. Wir wissen also nicht von Anfang an, wann die nächste Seite beginnt, wie groß der Rand, das Papier, die Schriftgröße wird, ganz abgesehen von der Schriftart. Sie können sich vorstellen, diese ganzen Parameter dynamisch in solch eine `Print`-Routine einzubauen, erfordert etwas mehr Aufwand, als nur vorgefertigte Klassen aufzurufen.

Um nur einen kleinen Einblick in dieses Szenario zu geben, möchte ich Ihnen ein kleines Beispiel zeigen, wie solch ein mehrseitiger Ausdruck realisiert werden könnte. Zu diesem Vorhaben bedienen wir uns einer Helferklasse, die ich `JuergensDokument` nenne und von `PrintDocument` ableite. Der Text dieses Dokuments soll dabei in einem Textarray gespeichert werden, die aktuelle Seiten- und Zeilennummer in einer `Integer`-Variablen.

In Listing 12.44 finden Sie den Programmcode für diese Klasse.

Listing 12.44
Helperklasse für PrintDocument

```
Public Class JuergensDokument
  Inherits Printing.PrintDocument

  Private mText() As String
  Private mSeitennummer As Integer
  Private mZeilennummer As Integer

  Public Property Text() As String()
    Get
      Return mText
    End Get
    Set(ByVal value As String())
      mText = value
    End Set
  End Property

  Public Property Seitennummer() As Integer
    Get
      Return mSeitennummer
    End Get
    Set(ByVal value As Integer)
       mSeitennummer = value
    End Set
  End Property

  Public Property Zeilennummer() As Integer
    Get
      Return mZeilennummer
    End Get
    Set(ByVal value As Integer)
      mZeilennummer = value
    End Set
  End Property

  Public Sub New(ByVal text() As String)
    Me.Text = text
  End Sub

End Class
```

In Listing 12.45 finden Sie den Programmcode für das Formular, das unser `PrintDocument` enthält.

Listing 12.45
Aufruf und Konfiguration für PrintDocument

```
Imports System.Drawing.Printing
Public Class Form1
  Private pd As JuergensDokument
```

Listing 12.45 (Forts.) Aufruf und Konfiguration für PrintDocument

```
Private Sub Form1_Load(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles Me.Load

   Dim zuDruckenderText(200) As String
   For i As Integer = 0 To 200
     zuDruckenderText(i) = i.ToString()
     zuDruckenderText(i) &= _
     " Visual Basic 2010  von Jürgen Kotz " & _
     "Addison-Wesley, München"
   Next
   pd = New JuergensDokument(zuDruckenderText)
   AddHandler pd.PrintPage, AddressOf printme
End Sub

Private Sub printme(ByVal sender As Object, _
   ByVal e As PrintPageEventArgs)

  pd = CType(sender, JuergensDokument)

  Dim f As New Font("Arial", 11)
  Dim Zeilenhoehe As Single = f.GetHeight(e.Graphics)
  Dim x As Single = e.MarginBounds.Left
  Dim y As Single = e.MarginBounds.Top
  pd.Seitennummer += 1
  Do
    e.Graphics.DrawString(pd.Text( _
      pd.Zeilennummer), f, Brushes.Black, x, y)
    pd.Zeilennummer += 1
    y += Zeilenhoehe
  Loop Until (y + Zeilenhoehe) > e.MarginBounds.Bottom Or
    pd.Zeilennummer > pd.Text.GetUpperBound(0)

  If pd.Zeilennummer < pd.Text.GetUpperBound(0) Then
    e.HasMorePages = True
  Else
    pd.Zeilennummer = 0
  End If
End Sub

Private Sub SeitenansichtToolStripMenuItem_Click _
  (ByVal sender As Object, ByVal e As System.EventArgs) _
   Handles SeitenansichtToolStripMenuItem.Click

   Dim PrintPreviewDialog1 As New PrintPreviewDialog
   PrintPreviewDialog1.Document = pd
   PrintPreviewDialog1.ShowDialog()
End Sub

Private Sub SeiteneinstellungenToolStripMenuItem_Click _
  (ByVal sender As Object, ByVal e As System.EventArgs) _
   Handles SeiteneinstellungenToolStripMenuItem.Click
```

Listing 12.45 (Forts.)
Aufruf und Konfiguration für PrintDocument

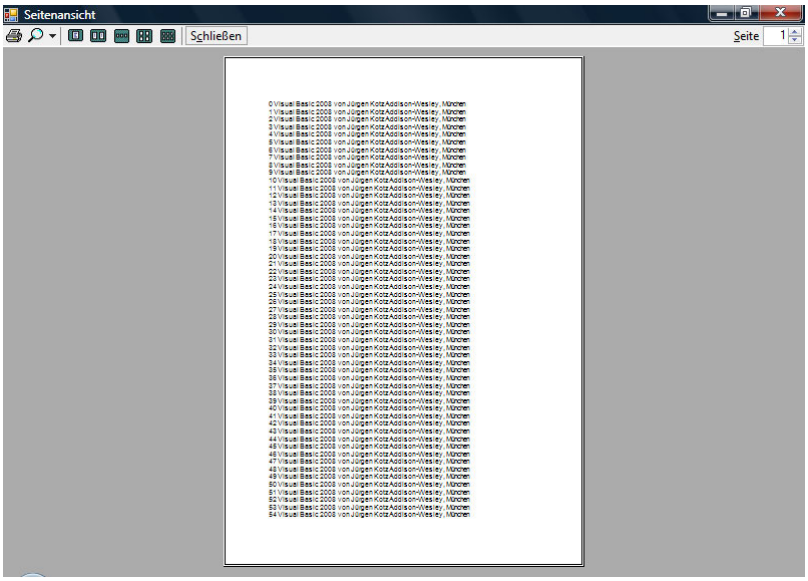
```
    Dim PageSetupDialog1 = New PageSetupDialog()
    PageSetupDialog1.Document = pd
    PageSetupDialog1.AllowMargins = True
    If PageSetupDialog1.ShowDialog() = _
           Windows.Forms.DialogResult.OK Then

      pd = PageSetupDialog1.Document
    End If
  End Sub

  Private Sub DruckenToolStripMenuItem_Click _
   (ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles DruckenToolStripMenuItem.Click

   Dim prd As New PrintDialog()
   prd.Document = pd
   prd.AllowPrintToFile = True
   prd.AllowSelection = True
   prd.AllowSomePages = True
   prd.ShowHelp = True
   If (prd.ShowDialog() = Windows.Forms.DialogResult.OK) Then
     pd = prd.Document
   End If
 End Sub
End Class
```

In der `Print`-Routine erhalten wir das zu druckende `PrintDocument`, indem wir das `sender`-Objekt in seinen ursprünglichen Objekttyp zurückwandeln. Nachdem wir die Höhe der Schrift und die Randkoordinaten des Blatts als Variablen zugewiesen haben, gehen wir das Textarray unserer `JuergensDokument`-Instanz in einer `Do Loop`-Schleife durch, bis wir den unteren Rahmen des Blatts erreicht haben oder kein Text mehr in unserer Speichervariablen vorhanden ist. Dabei geben wir bei jedem Schleifendurchgang die gerade aktuelle Zeile auf dem Drucker aus, rücken eine Zeile in dem Array weiter, indem wir die Zeilennummer inkrementieren, und setzen den *Schreibkopf* in die nächste Zeile, was bedeutet, dass wir die Y-Koordinate um die Zeilenhöhe erhöhen, die wir vorher anhand der Fontklasse ermittelt haben. Falls wir nun am unteren Rand angelangt sind, fragen wir in einer `If`-Bedingung noch ab, ob wir uns schon am Ende des Zeichenarrays befinden. Falls dies nicht der Fall ist, setzen wir die `HasMorePages`-Eigenschaft auf `True`, womit die `printme()`-Methode erneut durchlaufen wird.

Schließlich wollen wir uns in Abbildung 12.36 noch die Druckvorschau unseres Dokuments anschauen.

Wie Sie sicher festgestellt haben, war das sehr viel Handarbeit. In den beiden folgenden Abschnitten zeige ich Ihnen komfortable **Reportingtools**, die Ihnen sehr viel dieser Handarbeit abnehmen.

Abbildung 12.36
Seitenansicht des erzeugten PrintDocument



12.6.2 ReportViewer-Steuerelement

Das `ReportViewer`-Steuerelement von Microsoft bietet einige Funktionalitäten, was man schon an den 30 Eigenschaften erkennen kann, die durch ihre Namen verraten, dass man durch sie hauptsächlich das Aussehen verändern und das Aus- beziehungsweise Einblenden von Berichtswerkzeugen erreichen kann.

Das `ReportViewer`-Steuerelement wurde in Visual Studio 2005 eingeführt und befindet sich im Namespace `Microsoft.Reporting.WinForms`. Es gibt auch eine Version des Controls für ASP.NET, die sehr ähnlich wie die hier vorgestellte funktioniert.

Wie Sie am Namespace erkennen können, ist das `ReportViewer`-Steuerelement kein Bestandteil des .NET Frameworks. Es ist zwar in Visual Studio 2010 integriert und es wird auch mit ausgeliefert (nicht in der Express-Version), jedoch ist es eine separate Bibliothek, die bei Bedarf lizenzfrei von der Microsoft-Seite heruntergeladen werden kann.

Achtung

Bei einer Installation auf einem Client muss der `ReportViewer` extra installiert werden, da er nicht Bestandteil des Frameworks ist.

Mit dem `ReportViewer` ist eine einfache Erstellung von Berichten inklusive Datenbindung möglich.

Das Control bietet dabei verschiedenste Möglichkeiten zum Filtern, Sortieren, Gruppieren und Aggregieren von Daten. Es stehen verschiedene Layoutmöglichkeiten (Kreuztabellen, Charts, Tabellen) mit zahlreichen Formatierungsmöglichkeiten, auch programmatisch ansteuerbar, zur Datenpräsentation zur Verfügung.

Außerdem stehen zwei Exportformate für *Excel*- und *pdf*-Dateien zur Verfügung.

Der `ReportViewer` arbeitet in zwei unterschiedlichen `ProcessingModes`:

- Remote (oder auch Servermode)

  Bei diesem Modus werden die Microsoft SQL Server 2005/2008 Reporting Services verwendet. Die Reportdefinition (RDL) wird innerhalb einer SQL Server-Datenbank gespeichert. Die Reporting Services sind dabei für die Generierung der Daten sowie das Layout zuständig. Der gesamte Bericht, inklusive Daten und Layout, wird dabei vom Server übergeben. Die Kommunikation mit den Reporting Services erfolgt über Web-Services, deren Adresse mittels der Property `ReportServerUrl` angegeben wird. Die wichtigsten Eigenschaften werden über die Eigenschaften zu `ServerReport` gesetzt.

- Local (oder Clientmode)

  Bei diesem Modul liegt die gesamte Reportdefinition (RDLC) komplett auf dem Client. Der Client ist auch verantwortlich für die Generierung und Bereitstellung der anzuzeigenden Daten innerhalb der Applikation. Mittels der Property `ReportEmbeddedResource` wird ein Verweis auf die lokale Berichtsdefinition gesetzt, die im Steuerelement angezeigt werden soll. Die wichtigsten Eigenschaften werden über die Eigenschaft `LocalReport` gesetzt. Der Report wird innerhalb der Applikation als Embedded Resource gespeichert. Es besteht auch die Möglichkeit, ihn als eigenständige *rdlc*-Datei zu speichern.

Schauen wir uns die Vorgehensweise bei der Erstellung eines Reports an.

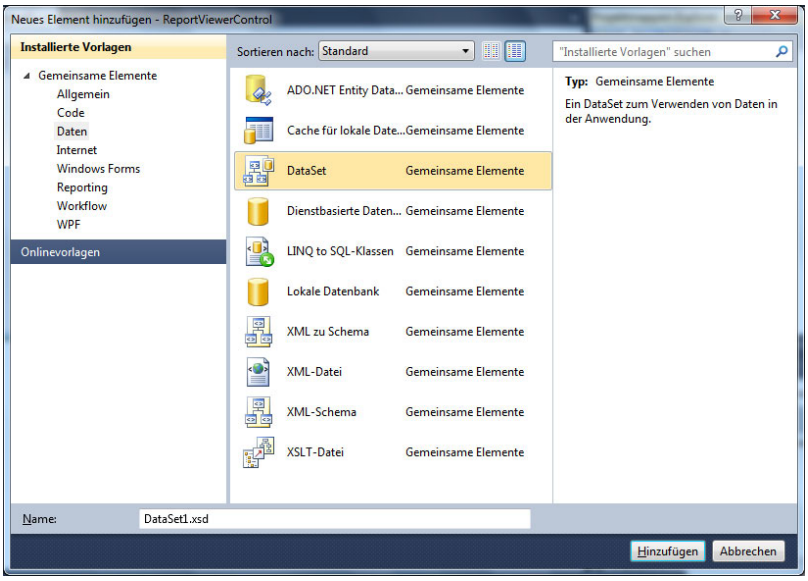
Zuerst ein kleines Rezept, wie man generell einen Bericht mit dieser Komponente erstellen kann:

1. Neues Item *Report.rdlc* zum Projekt hinzufügen
2. Datenquelle einbinden, zum Beispiel durch ein Dataset
3. Im Berichtdesigner ein Report-Item von der Toolbox sowie eine Tabelle auf den Body des Berichts ziehen
4. Felder aus der Datenquelle auf die gewünschten Sektionen des Berichts ziehen
5. Eventuelle Gruppierungen, Formate oder sonstige Regeln hinzufügen
6. Ein `ReportViewer`-Control auf eine `WindowsForms` ziehen und dieser den Bericht zuordnen
7. Mit F5 die Applikation starten und sich freuen, falls alles geklappt hat

Aber gehen wir ein Beispiel exemplarisch Schritt für Schritt durch.

Zuerst fügen wir als neues Item ein `DataSet` zu unserem Projekt hinzu (siehe Abbildung 12.37) und füllen es anschließend, indem wir auf die Design-Oberfläche einen `TableAdapter` ziehen. Als Datenquelle benutze ich dabei die `Nordwind`-Beispieldatenbank und hole mir die Tabellen *Artikel*, *Bestelldetails*, *Bestellungen* und *Kunden*.

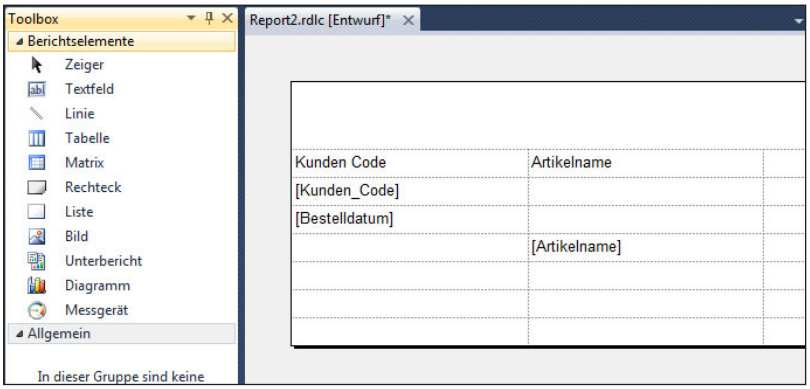
Abbildung 12.37
DataSet zum Projekt hinzufügen



Anschließend ziehen Sie eine Tabelle aus der Toolbox auf die Berichtoberfläche (siehe Abbildung 12.38).

Wie Sie erkennen können, unterteilt sich der Bericht in einen *Header*-, einen *Detail*- und einen *Footer*-Bereich. Sie können schon loslegen, indem Sie Felder aus der `DataTable` in die gewünschten Sektionen ziehen. Doch zuerst müssen wir uns überlegen, was wir überhaupt darstellen möchten. Nun – ich möchte wissen, welche Kunden welche Artikel an welchem Tag gekauft haben. Und natürlich wird nach den Kunden sortiert, die am meisten gekauft haben.

Abbildung 12.38
ReportViewer – Berichtsgenerierung



Um alle Bestellungen auszugeben, ziehen wir zunächst das Feld *Artikelname* in die DETAILS-Region, gefolgt von dem Feld *Einzelpreis*, das wir rechts daneben setzen. Um diese einzelnen Datensätze jetzt noch nach den Kunden zu

gruppieren, klicken wir mit der rechten Maustaste auf den linken Rand des Reports und wählen Gruppe hinzufügen. Es erscheint der Dialog, den Sie in Abbildung 12.39 sehen.

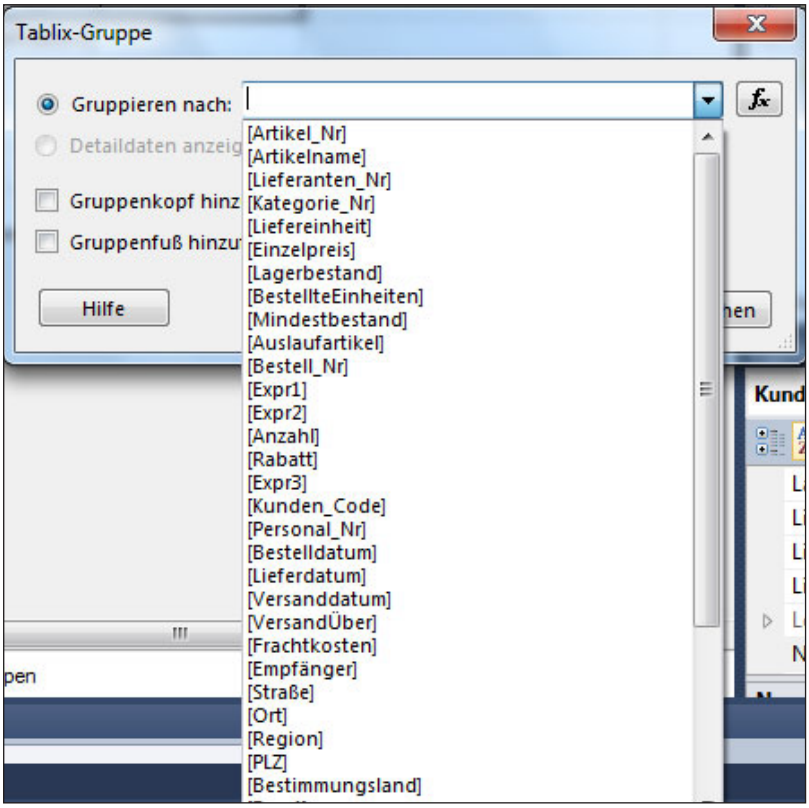
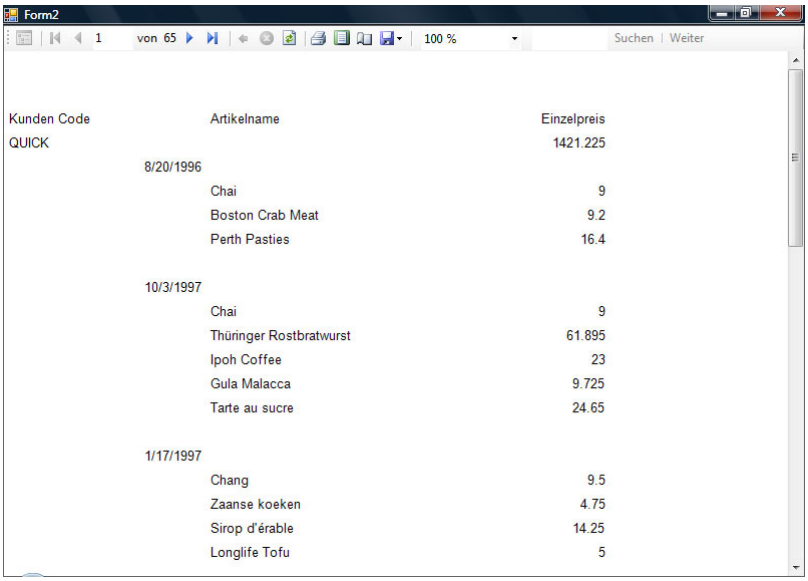


Abbildung 12.39
Gruppe zum Bericht hinzufügen

Bei Gruppieren nach geben wir an, nach welchem Feld wir gruppieren möchten, wählen den Wert des Felds *Kunden-Code* und bestätigen diesen. Diesen Vorgang wiederholen wir gleich noch einmal, wobei wir statt des Felds *Kunden-Code* das Feld *Bestelldatum* auswählen. Jetzt dürfte ihr Bericht ungefähr das Aussehen haben, wie es bereits die Abbildung 12.38 weiter vorn zeigt.

Um nun noch die Gesamtsumme jedes Kunden auszugeben, ziehen wir einfach per Drag&Drop das Feld *Einzelpreis* in die zweite Zeile der gleichnamigen Spalte. Um unseren Minimalbericht jetzt in Aktion zu erleben, müssen wir noch ein paar Schritte unternehmen. Dazu wechseln wir zu einem Windows-Formular, ziehen ein `ReportViewer`-Control auf dessen Oberfläche und weisen diesem mittels des Smart Tags unseren gerade erstellten Report zu. Noch eine passende `Dock`-Eigenschaft (zum Beispiel `Fill`) setzen und das war's – mit der Taste F5 können wir nun den Report betrachten.

Abbildung 12.40
Kundenreport



Ihnen werden wohl noch ein paar Unschönheiten in Abbildung 12.40 aufgefallen sein, zum Beispiel die Formatierung des Bestelldatums. Um dieses in unserem gewohnten Datumsformat darstellen zu können, öffnen wir wieder den Bericht-Designer und gehen zu den Eigenschaften des Felds. Auf der Registerkarte FORMAT können wir dann unser gewohntes Datumsformat auswählen (siehe Abbildung 12.41).

Für komplexere Ausdrücke können Sie dabei auch mit den Schaltflächen `fx` einen mächtigen Ausdrucks-Editor aufrufen.

Wir können hier sicherlich nicht jede Einzelheit dieses Editors besprechen – das ist nach meiner Meinung auch nicht nötig. Denn wenn Sie sich erst einmal mit dem Ausdrucks-Editor angefreundet haben, dann haben Sie ein mächtiges Werkzeug zur Hand, das Sie am Anfang wohl nur als kleine Hilfestellung betrachtet haben. Wenn Sie ein beliebiges Feld auf dem Bericht markieren und sich dann die Eigenschaften dazu etwas genauer anschauen, so werden Sie feststellen, dass Sie fast jeder Eigenschaft anhand eines Ausdrucks dynamisch einen Wert zuweisen können. Dazu ein kurzes Beispiel: Nehmen wir an, wir wollen den Einzelpreis einer Bestellung mit roter Schriftfarbe ausgeben, wenn der Wert größer als 100 ist, ansonsten Schwarz. Was machen wir also? Wir markieren den Einzelpreis in der DETAILS-Region, wählen im Eigenschaftenfenster die Property *Color* aus und in der Auswahlliste den Eintrag <AUSDRUCK...>.

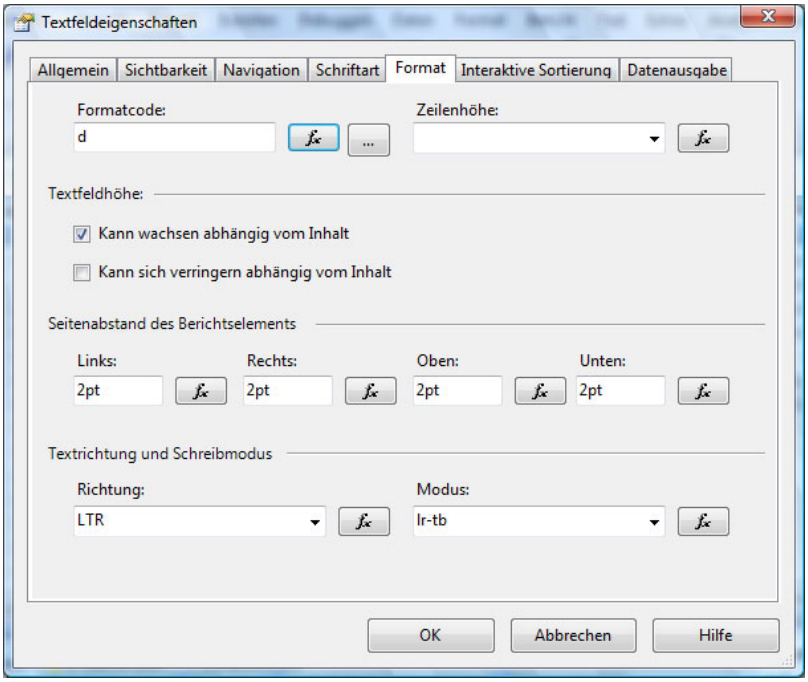


Abbildung 12.41
Feldeigenschaften

Es erscheint daraufhin der Dialog, der in Abbildung 12.42 zu sehen ist.

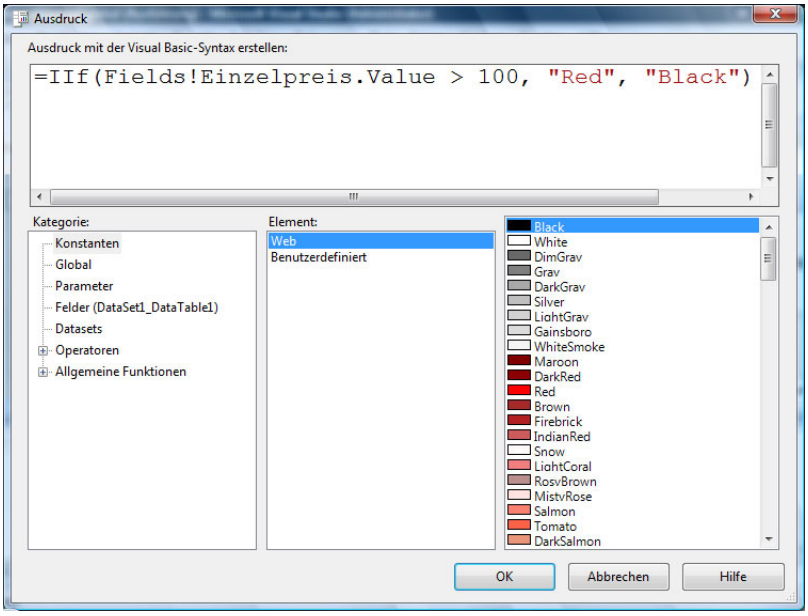


Abbildung 12.42
Ausdrucks-Editor

Dies soll nur die Mächtigkeit dieses Editors zum Ausdruck bringen. Sie können praktisch jedes Detail in Ihrem Report dynamisch gestalten. Dass der Editor auch noch IntelliSense-Unterstützung bietet, macht die ganze Sache noch komfortabler.

Weitere Verbesserungen am Layout bleiben gerne Ihnen überlassen.

12.6.3 Crystal Reports

Seit 1993 ist **Crystal Reports** Teil der Professional Version des Visual Studio und somit direkt in die Entwicklungsumgebung integriert. Mit dem in Visual Studio zur Verfügung gestellten Crystal Reports kann man komplexe und professionelle Reports mit Hilfe des integrierten GUI-Designers erstellen.

Crystal Reports ist dabei ein noch mächtigeres Tool als der gerade vorgestellte `ReportViewer`. Und ich kann Ihnen versprechen, dass es sich lohnen wird, einen kleinen Blick darauf zu werfen. Wir alle wissen, dass bestimmte Aufgaben beim Reporting immer wieder gewünscht sind, und warum sollten wir diese immer wieder implementieren, wenn es doch ein im Studio integriertes Control gibt, das sämtliche Datenquellen unterstützt? Crystal kann dabei nicht nur für Windows-Applikationen, sondern auch für Webapplikationen oder als Web-Service genutzt werden.

Mit Visual Studio 2010 wird die Version *Crystal Report 13* ausgeliefert, das ein sehr mächtiges Reportingtool mit sehr hoher Funktionalität und Programmierbarkeit ist. Während der Laufzeit sind die Berichte mit Crystal-Formeln sehr stark konfigurierbar. Dabei gibt es für die Crystal-Formen zwei Syntaxvarianten – Crystal-Syntax und VB-Syntax.

Crystal Reports unterstützt Datenbindung aus sehr vielen unterschiedlichen Datenquellen und die Reports sind sehr einfach und intuitiv zu erstellen, auch wenn man sich manchmal den Editor etwas komfortabler wünscht (zum Beispiel bei der Mehrfachmarkierung).

Die Berichtsdefinition wird in *rpt*-Files gespeichert und wird der Applikation standardmäßig als Embedded Resource hinzugefügt.

Auch hier will ich mit Ihnen exemplarisch einen Report erstellen.

Starten Sie ein neues Projekt und wählen Sie dabei Crystal Reports-Anwendung aus (siehe Abbildung 12.43).

Daraufhin zeigt sich ein Dialog, über den Sie den Berichtsassistenten starten können. Im nächsten Schritt erscheint eine Auswahl der anzuzeigenden Daten im Bericht. Ich wähle dabei die Kunden-Tabelle aus der Nordwind-Datenbank aus, wie Sie in Abbildung 12.44 sehen. Im folgenden Dialog können Sie die Felder auswählen, die Sie im Bericht anzeigen wollen, und daraufhin nach bestimmten Feldern gruppieren. Dabei wählen Sie alle Felder aus und für die Gruppierung wählen Sie das Feld *Kunden.Land* aus. Danach können Sie mit der Schaltfläche Fertig stellen die Berichtsdefinition abschließen.

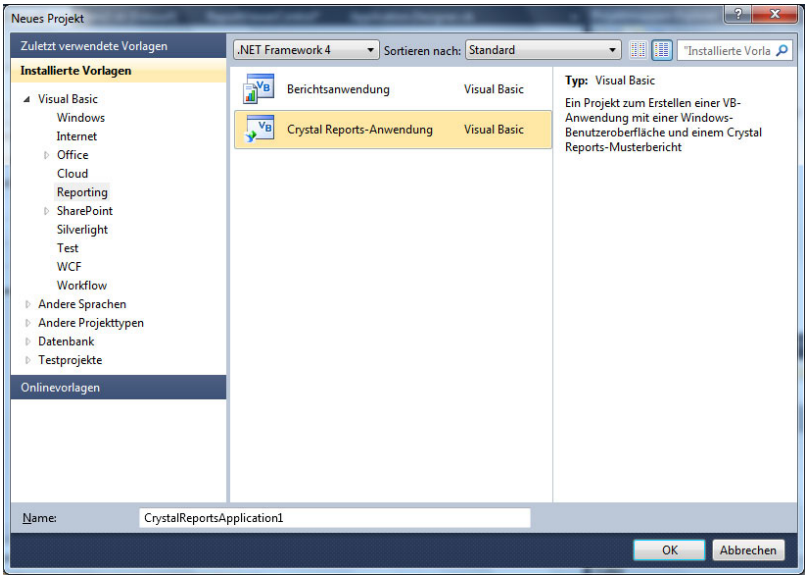


Abbildung 12.43
CrystalReports-Anwendung

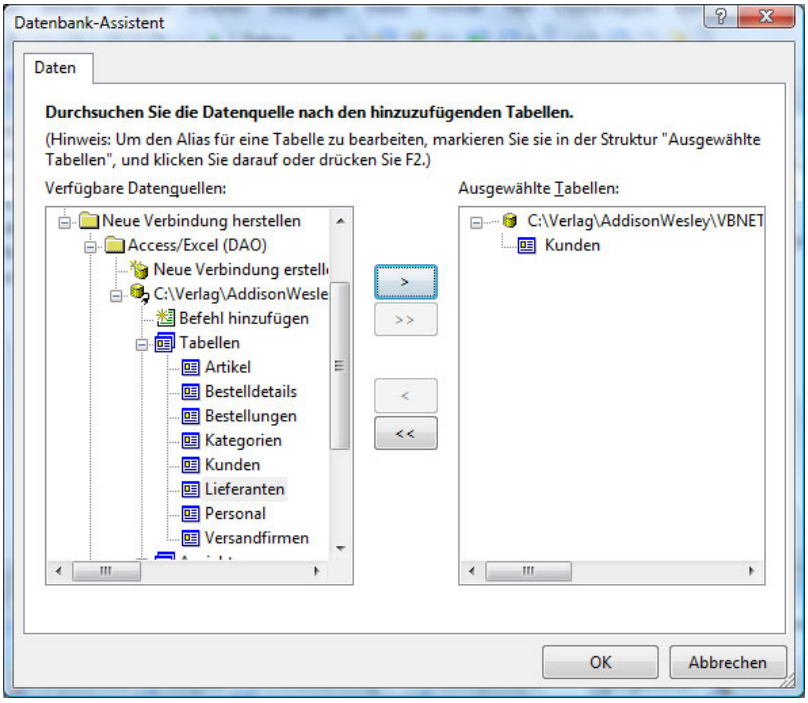


Abbildung 12.44
Datenauswahl

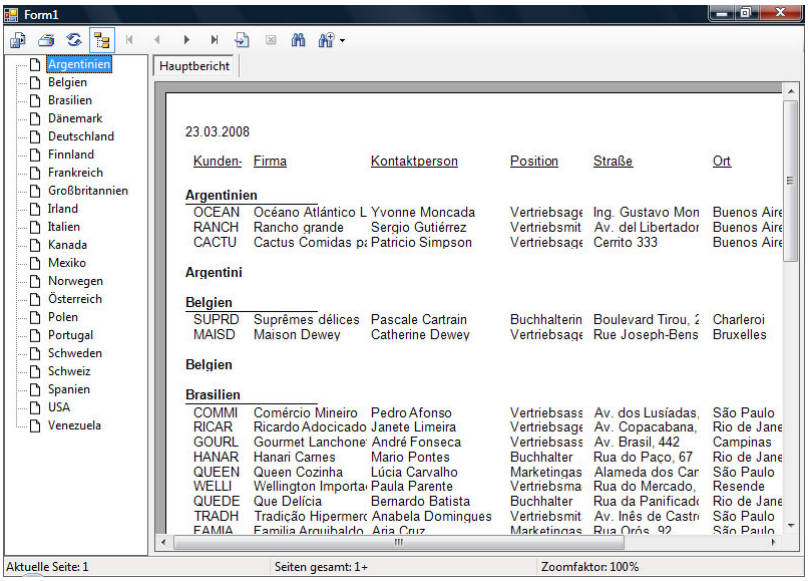
Im Editor können Sie anschließend noch Formatierungen vornehmen und Formeln hinterlegen, sehr ähnlich, wie es beim `ReportViewer` vorgestellt wurde. Die Möglichkeiten, die Sie hier haben, sind enorm, von Formelfeldern (inkl. Formeleditor), Parameterfeldern, Spezialfeldern über sonstige Assistenten. Sie können auch hier jeden einzelnen Punkt dynamisch gestalten.

Wenn Sie mit der Berichtsdefinition fertig sind, werden Sie sehen, dass in Ihrem Projekt bereits ein Formular angelegt wurde, auf dem ein `CrystalReportViewer`-Control platziert ist.

Sie können direkt das Projekt starten und sehen Ihren Report wie in Abbildung 12.45 gezeigt.

Das war hier nur eine sehr kurze Einführung in Crystal Reports, aber Sie dürfen mir glauben, dass wir hier nur an den Oberflächen der Funktionalität gekratzt haben. In Crystal Reports gibt es noch eine Menge zu erforschen und wenn Sie komplexe Reports erstellen müssen, dann lohnt sich ein genauerer Blick auf dieses Reportingtool allemal.

Abbildung 12.45 Ausgabe eines CrystalReports



13

Interoperabilität

Auch wenn wir es gerne verdrängen, gibt es eine Zeit vor .NET, in der wir Programmcode geschrieben haben. In diesem Kapitel will ich darauf eingehen, wie das Zusammenspiel mit COM-Komponenten funktioniert und wie Sie **Win32**-API-Funktionen aufrufen können.

13.1 Win32-Aufrufe

Obwohl das .NET Framework sehr mächtig ist, gibt es doch noch ein paar Funktionen der Windows API, die nicht im Framework implementiert sind. Wie Sie diese aus Ihrer .NET-Applikation heraus aufrufen können, will ich Ihnen hier kurz erläutern.

Seien Sie sich jedoch bewusst, dass Aufrufe an die Win32-API **unmanaged** Codeaufrufe darstellen. Das bedeutet, dass Sie unter Umständen mit nicht .NET-konformen Datentypen in Berührung kommen (und seien Sie sich gewiss, dass ein String in .NET etwas anderes ist als ein String in einer API-Funktion) und Sie für den API-Aufruf auch das Recht besitzen müssen, **unmanaged Code** aufzurufen.

Tipp

Verwenden Sie Win32-API-Aufrufe wirklich nur an den Stellen, wo es nicht anders geht.

Um einen API-Aufruf aus Visual Basic 10 heraus durchzuführen, benötigen Sie einen speziellen Dienst – **PInvoke** (PlatformInvocation). Es gibt in Visual Basic 10 zwei Möglichkeiten, PInvoke zu verwenden.

13.1.1 PInvoke mittels Declare

Mittels des Schlüsselworts `Declare` können wir API-Funktionen innerhalb unserer Applikationen deklarieren und dann innerhalb dieser Applikation auch nutzen.

Im folgenden Beispiel will ich die Funktion `FlashWindow` verwenden, welche die Titelleiste eines Formulars blinken lässt.

Dazu verwende ich folgende Deklaration:

```
Private Auto Declare Function FlashWindow _
    Lib "user32" Alias "FlashWindow" _
    (ByVal hwnd As Integer, _
    ByVal bInvert As Integer) As Integer
```

Durch diese Definition haben wir aus der Windows-Bibliothek *user32.dll* die Funktion `FlashWindow` definiert, die zum Aufruf zwei `Integer`-Variablen benötigt und deren Rückgabewert vom Typ `Integer` ist.

Die beiden weiteren Schlüsselwörter `Auto` und `Alias` haben in diesem Beispiel keine große Bedeutung, da hier keine Stringverarbeitung stattfindet. Nichtsdestotrotz will ich sie kurz erläutern.

`Auto` bedeutet, dass die Zeichenfolgen nach den Regeln der Laufzeitumgebung konvertiert werden. Alternativen zu `Auto` sind `Ansi` oder `Unicode`, wenn Sie eine spezielle Zeichenkonvertierung vornehmen wollen.

Mit dem `Alias`-Schlüsselwort verweisen Sie auf den tatsächlichen Methodennamen innerhalb der Windows-Bibliothek. Dadurch können Sie den hinter `Function` angegebenen Funktionsnamen frei wählen, falls er in Konflikt mit bestehenden Methoden stehen würde.

Nachdem ich diese Funktion deklariert habe, rufe ich sie innerhalb eines `Timer`-Steuerelements auf, um mein Formular blinken zu lassen, wie in Listing 11.1 dargestellt.

An die Funktion wird als erster `Integer`-Wert das **Fensterhandle** des zu blinkenden Formulars übergeben und als zweiter Parameter eine Variable, die den Wert 0 oder 1 besitzt, damit es auch wirklich blinkt.

Listing 13.1 Timer-Ereignis für den Aufruf der API-Funktion FlashWindow

```
Private Sub Timer1_Tick (ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles Timer1.Tick

  If invert = 0 Then
    FlashWindow(Me.Handle, invert)
    invert = 1
  Else
    FlashWindow(Me.Handle, invert)
    invert = 0
  End If
End Sub
```

Info

Unter *http://www.freeware-archiv.de/ApiViewer-Programmieren.htm* finden Sie einen API-Viewer, der Ihnen die Deklaration der API-Funktionen größtenteils abnimmt. Der Link hat zumindest zum Zeitpunkt der Fertigstellung des Buchs noch funktioniert.

13.1.2 PInvoke mittels DLLImport

Im zweiten Beispiel zeige ich Ihnen die Verwendung von **PInvoke** mittels des Attributs `DLLImport`. Dabei will ich eine Funktion zeigen, die mit Strings arbeitet, damit Sie die Besonderheiten bei der Bearbeitung von *Strings* im Zusammenhang mit API-Aufrufen auch gesehen haben.

Damit Sie das Attribut `DLLImport` ansprechen können, müssen Sie zuerst den Namensraum `System.Runtime.InteropServices` einbinden.

```
Imports System.Runtime.InteropServices
```

Danach definieren wir die gewünschte API-Funktion mit den benötigten Parametern wie eine herkömmliche .NET-Methode mit einem leeren Rumpf. Die einzige Besonderheit ist, dass die Funktion statisch, also mit dem Schlüsselwort `Shared` versehen, sein muss.

```
Public Shared Function GetUsername _
  (ByVal lpbuffer As String,   ByRef nsize As Integer) As Integer
End Function
```

Somit haben wir im ersten Schritt eine leere Funktion implementiert, die eine identische Signatur mit unserer gewünschten API-Funktion besitzt.

Jetzt müssen wir mit dem Attribut `DLLImport` noch dafür sorgen, dass bei einem Funktionsaufruf dieser Aufruf auch an die Windows-API weitergeleitet wird.

Dazu setzen wir das Attribut folgendermaßen für unsere leere Funktion:

```
<DllImport("advapi32.dll", CharSet:=CharSet.Auto)> _
Public Shared Function GetUsername _
  (ByVal lpbuffer As String, ByRef nsize As Integer) As Integer
End Function
```

Dabei geben wir als ersten Parameter den Windows-Bibliotheksnamen bekannt. Alle weiteren Parameter sind optionale Parameter, das heißt, sie können, müssen aber nicht angegeben werden. Da wir hier eine Vielzahl von Parametern haben, von denen wir nur einen nutzen wollen, verwenden wir hier **benannte Parameter**. Dabei gibt man den Parameternamen an und mit `:=` weisen wir diesem einen Wert zu, unabhängig von der Reihenfolge der Parameterliste.

Der einzige optionale Parameter, den wir hier nutzen, ist wieder die Angabe der zu konvertierenden Zeichenfolge. Dabei wählen wir wiederum den Wert `Auto`, so dass die Laufzeitumgebung die richtige Konvertierung für uns durchführt.

Jetzt müssen wir nur noch die Funktion aufrufen wie in Listing 13.2.

```
Dim username As String = Space(255)
GetUserName(username, 255)
MessageBox.Show(username)
```

Listing 13.2
Aufruf der API-Funktion GetUserName()

Wie Sie in diesem Beispiel sehen, wird der übergebene String vorinitialisiert, und zwar mit 255 Leerzeichen. Wenn wir das nicht tun, kann die API-Funktion keinen Wert eintragen, da tatsächlich die Speicheradresse übergeben wird. Und wenn der entsprechende Speicherplatz nicht groß genug ist, kann es auch zu sehr beträchtlichen und unangenehmen Beeinträchtigungen des Systems bis hin zum Absturz des gesamten Rechners kommen. Der zweite Parameter gibt dementsprechend die Größe des vorinitialisierten Strings an, dieser Wert sollte auch immer übereinstimmen.

Tipp

Wenn Sie eine API-Funktion nutzen, lesen Sie bitte auch die Dokumentation für diese API-Funktion, um sicherzugehen, dass alle Parameter passend übergeben werden. Oder noch besser: Suchen Sie zuerst in der .NET-Dokumentation, ob es im .NET Framework nicht doch eine passende Methode dafür gibt.

Zum Abschluss will ich noch eine API-Funktion vorstellen, die ich sehr gerne verwende. `SetParent()` bietet die Möglichkeit, ein Formular innerhalb eines Containers eines anderen Formulars zu laden. Dies ist vor allem bei Anwendungen im Explorer-Stil eine häufig genutzte Technologie.

Definieren Sie dazu die `SetParent()`-Methode entweder mit dem Schlüsselwort `Declare Function` oder mit dem `DLLImport`-Attribut.

Listing 13.3 SetParent()-Definition mit Declare Function

```
Private Declare Function SetParent
   Lib "user32" Alias "SetParent"(ByVal hWndChild As IntPtr, _
   ByVal hWndNewParent As IntPtr) As Integer
```

Auf ein Formular habe ich einen `SplitContainer` gezogen und einen `Button`, um ein anderes Formular innerhalb eines `Panel` des `SplitContainer` zu laden. Außerdem habe ich ein neues Form zum Projekt hinzugefügt, wobei ich die Vorlage *LoginForm* genommen habe, um mir die Arbeit eines Formularlayouts zu sparen.

Zum Laden des Formulars habe ich zum Ereignishandler des `Button` den Code in Listing 13.4 implementiert.

Listing 13.4 Einsatz von SetParent()

```
Private Sub btnLadeForm_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnLadeForm.Click

  Dim dialog As New LoginForm1
  dialog.Top = 0
  dialog.Left = 0
  dialog.WindowState = FormWindowState.Maximized
  SetParent(dialog.Handle, SplitContainer1.Panel2.Handle)
  dialog.Show()
End Sub
```

In Listing 13.4 sehen Sie, dass ich das Formular ganz normal instanziiere, das Formular maximiere und so darstelle, dass es in der linken oberen Ecke des Containers angezeigt wird. Bevor ich das Formular mit `Show()` anzeige, weise ich mit der Methode `SetParent()` das `Panel` des `SplitContainer` als `Parent` zu. Das Formular wird schließlich innerhalb dieses `Panel` geladen, wie Sie in Abbildung 13.1 sehen.

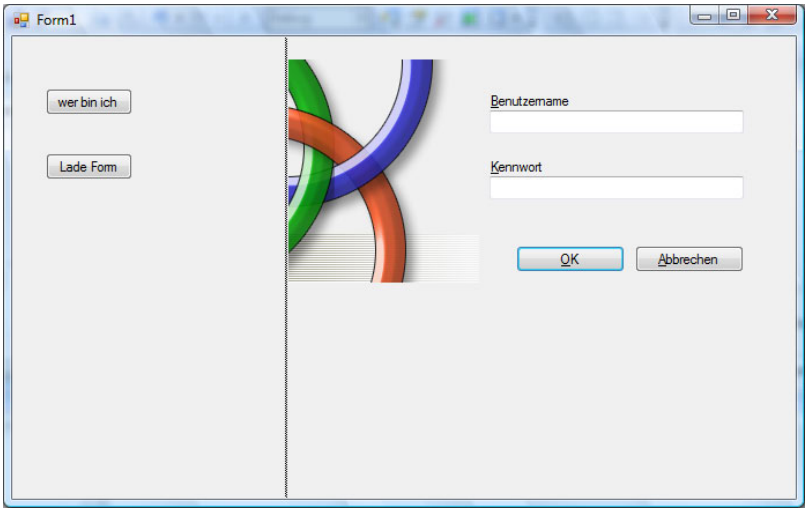


Abbildung 13.1
Geladenes Formular innerhalb eines Panel

13.2 COM-Interop

Unter COM-Interop versteht man die Kommunikation zwischen einer .NET-Komponente und einer COM-Komponente. Da beide Komponenten nicht direkt miteinander kommunizieren können, werden **Wrapper** benötigt, welche die Aufrufe und Rückgabewerte jeweils für die andere Seite verständlich umsetzen. Dabei gibt es zwei unterschiedliche Wrapper:

- Runtime Callable Wrapper (RCW)

  Dieser Wrapper wird eingesetzt, wenn aus einer .NET-Assembly eine COM-Funktionalität aufgerufen wird.

- COM Callable Wrapper (CCW)

  Dieser Wrapper wird eingesetzt, wenn aus einer COM-Komponente Funktionalität aus einer .NET-Assembly aufgerufen wird.

Diese beiden Kommunikationsmöglichkeiten wollen wir hier betrachten. Zuerst wollen wir uns aber noch kurz die wesentlichsten technologischen Unterschiede zwischen einer .NET-Assembly und einer COM-Komponente ins Gedächtnis rufen.

1. Lebensdauer

   Die Lebensdauer von Objekten wird unter .NET von der Laufzeitumgebung verwaltet, bei COM müssen die Objekte ihre Lebenszeit selbst verwalten (jedes Objekt besitzt einen eigenen Referenzzähler).

2. Schnittstellendefinitionen

   COM-Objekte stellen Ihre Schnittstellenbeschreibungen über Interfaces zur Verfügung. Die Interfaces geben auf Anfrage einen Schnittstellenzeiger zurück. Bei .NET-Assemblies können die Schnittstellen über Reflections abgefragt werden.

3. Speicherverwaltung

   Unter .NET ist es möglich, dass Objekte im Speicher bewegt werden, die Adressen auf diese Objekte werden automatisch angepasst. Unter COM stehen Objekte im Speicher an einer festen Position, eine Verschiebung dieser Speicherbereiche ist nicht möglich.

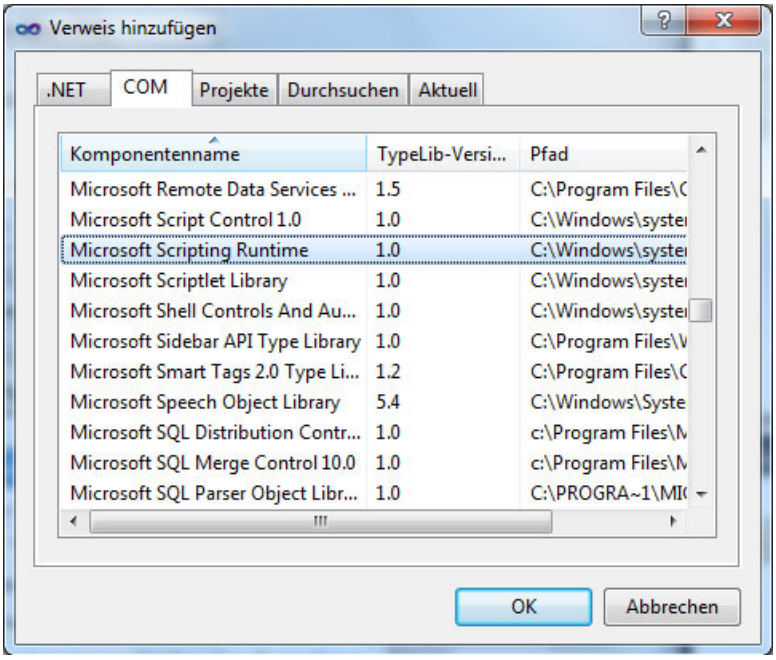
13.2.1 Runtime Callable Wrapper

Einen Runtime Callable Wrapper (RCW) benötigen Sie immer dann, wenn Sie aus Ihrer .NET-Applikation heraus eine COM-Komponente oder auch ein ActiveX-Steuerelement benutzen wollen.

Diese Funktionalität wird man noch öfter brauchen, da es sehr viel »alten« COM-Code gibt, der nicht innerhalb von kürzester Zeit auf die neue Technologie portiert werden kann. Dieser Weg funktioniert relativ einfach und problemlos.

Sie müssen sich nur vergewissern, dass die entsprechende COM-Komponente auf dem Rechner installiert und registriert ist. Sie können dann Ihrem Projekt einen Verweis auf diese COM-Bibliothek hinzufügen (siehe Abbildung 13.2).

Abbildung 13.2
COM-Verweis hinzufügen



Für die COM-Bibliotheken gibt es ein eigenes Register zum Auswählen der entsprechenden Komponenten, wie Sie in Abbildung 13.2 sehen.

Nachdem Sie die gewünschte COM-Bibliothek gewählt haben, wird eine generische Wrapper-Bibliothek in das Projektverzeichnis gelegt. Dieser Wrapper besitzt alle Schnittstellen der Originalbibliothek. Sie programmieren aus

Ihrem .NET-Projekt heraus gegen diese Wrapper-Bibliothek, welche die entsprechenden Aufrufe dann in einer COM-kompatiblen Syntax mit COM-kompatiblen Datentypen an die ursprüngliche COM-Bibliothek weiterleitet.

Die Wrapper heißen genauso wie die zugrunde liegende COM-Bibliothek mit dem Präfix `Interop`.

Sie merken faktisch gar nicht, dass die dahinter liegende Bibliothek keine kompatible .NET-Bibliothek ist. Der Wrapper nimmt Ihnen an dieser Stelle die ganze Sache ab.

Die Teile der Anwendung liegen auch in unterschiedlichen Speicherbereichen. Während die .NET-Assembly im verwalteten Speicher der Laufzeitumgebung liegt und somit auch vom Garbage Collector hinsichtlich der Speicherverwaltung gesteuert wird, liegt die COM-Komponente außerhalb des Verantwortungsbereichs der CLR in einem unverwalteten Bereich.

Für die Office-Applikationen werden schon vorkompilierte Interop-Bibliotheken mit ausgeliefert, da diese vom Umfang doch etwas größer sind und es sehr lange dauern würde, bis die Wrapper-Bibliotheken erstellt wären.

Achtung

Seien Sie sich jedoch bewusst, dass eine .NET-Applikation, die auf COM-Komponenten zugreift, bei der Installation wiederum die altbekannten Probleme wie DLL-Hell und Registry-Einträge mit sich bringt. Denn die COM-Komponenten müssen nun auf allen Clientrechnern sauber installiert und registriert werden.

13.2.2 COM Callable Wrapper

Einen COM Callable Wrapper (CCW) benötigen Sie immer dann, wenn Sie von einer COM-Anwendung heraus Funktionen einer .NET-Assembly nutzen wollen.

Bei einem Aufruf aus einer COM-Komponente heraus zu einer .NET-Assembly erstellt die CLR genau einen einzigen CCW, unabhängig davon, wie viele COM-Clients auf diesen zugreifen. Der CCW leitet dabei die Aufrufe von den COM-Clients an das .NET-Objekt weiter.

Außerdem verwaltet der CCW die Identität und die Lebensdauer des .NET-Objekts. Der CCW selbst funktioniert nach dem COM-Prinzip der Referenzzählung. Er führt Buch darüber, wie viele Clients gerade auf ihn verweisen. Ist der Referenzzähler irgendwann bei 0 angekommen, gibt der CCW den Verweis auf das .NET-Objekt frei, was dazu führt, falls keine anderen .NET-Objekte auf dieses Objekt referenzieren, dass der Garbage Collector beim nächsten Durchlauf das .NET-Objekt im Speicher wieder freigibt.

Das .NET-Objekt liegt dabei, wie gerade angesprochen, in der Verantwortung des Garbage Collectors und kann im Speicher verschoben werden. Der CCW hingegen liegt in einem Speicherbereich innerhalb des Heap, der nicht vom Garbage Collector verschoben wird, so dass die COM-Clients direkt auf diesen Wrapper zeigen können.

Was müssen Sie aber jetzt alles tun und beachten, damit Ihre .NET-Typen überhaupt aus COM heraus angesprochen werden können:

- Jeder Typ muss öffentlich sein (also als public definiert sein).
- Jeder Typ muss explizit ein Interface implementieren, in dem alle Member enthalten sind, die unter COM sichtbar sein sollen.
- Der Standardkonstrukor (ohne Parameter) muss öffentlich sein.
- Der Typ darf nicht als abstrakte Klasse definiert sein (kein MustInherit erlaubt).
- Sie können Interop-Attribute verwenden, um die Interoperabilität zu verbessern.
- Die .NET-Assembly muss mit dem Tool *regasm.exe* in der Registry registriert werden.
- Im COM-Projekt muss ein Verweis auf die .NET-Bibliothek gesetzt werden.
- Ein .NET-Typ muss im COM-Projekt über den Standardkonstruktor instanziiert werden.

Das sind die Grundregeln, die Sie beachten müssen.

Ich habe in der folgenden Beispielanwendung eine kleine Klasse geschrieben, in der wir die ganzen Regeln einmal in der Praxis umgesetzt sehen. Den Code der Klassenbibliothek sehen Sie in Listing 13.5.

Listing 13.5 Beispiel für eine COM-sichtbare Klasse

```
Public Interface ITestKlasse
    Function WerBistDu() As String
    Property MeinName() As String
End Interface
<Microsoft.VisualBasic.ComClass()>
Public Class TestKlasse
    Implements ITestKlasse

    Public Sub New()
    End Sub

    Private mName As String
    Public Property MeinName() As String _
        Implements ITestKlasse.MeinName

        Get
            Return mName
        End Get
        Set(ByVal value As String)
            mName = value
        End Set
    End Property

    Public Function WerBistDu() As String _
        Implements ITestKlasse.WerBistDu

        Return "Ich bin " & mName
    End Function
End Class
```

Wir haben hier eine Klasse `Testklasse`, die das Interface `ITestklasse` mit einer Eigenschaft und einer Methode implementiert.

Danach habe ich nur noch über Eigenschaften die oben angesprochenen `Interop`-Attribute gesetzt.

Die erste Eigenschaft war unter MyProjekt – Öffnen – Anwendung – Assembly-Informationen.... In den Assemblyinformationen habe ich ein Häkchen im Kontrollkästchen Assembly COM-sichtbar machen aktiviert (siehe Abbildung 13.3).

Dadurch wurde in der *AssemblyInfo.vb* folgender Eintrag hinzugefügt:

```
<Assembly: ComVisible(True)>
```

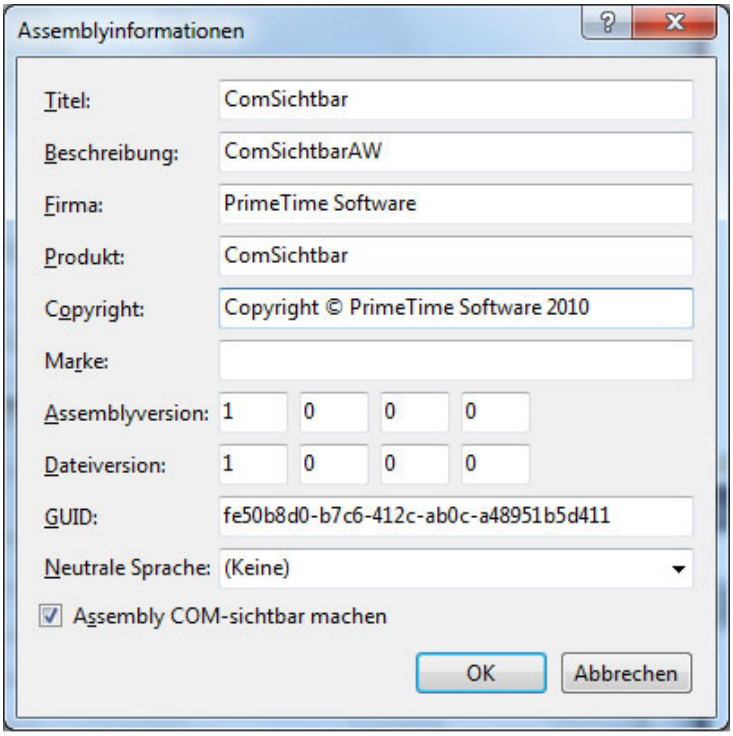


Abbildung 13.3
Assembly COM sichtbar machen

Danach habe ich vor die Klasse das Attribut `<Microsoft.VisualBasic.ComClass()>` gesetzt.

Als Nächstes will ich noch, dass sich die Bibliothek selbst in die Registry einträgt, ohne dass ich das Tool *regasm.exe* selber aufrufen muss.

Dazu habe ich unter MyProjekt – Öffnen – Kompilieren das Kontrollkästchen Für COM-Interop registrieren angeklickt (siehe Abbildung 13.4).

Und schon ist die Hauptarbeit erledigt, alle gewünschten Interop-Attribute wurden durch diese Aktionen automatisch gesetzt.

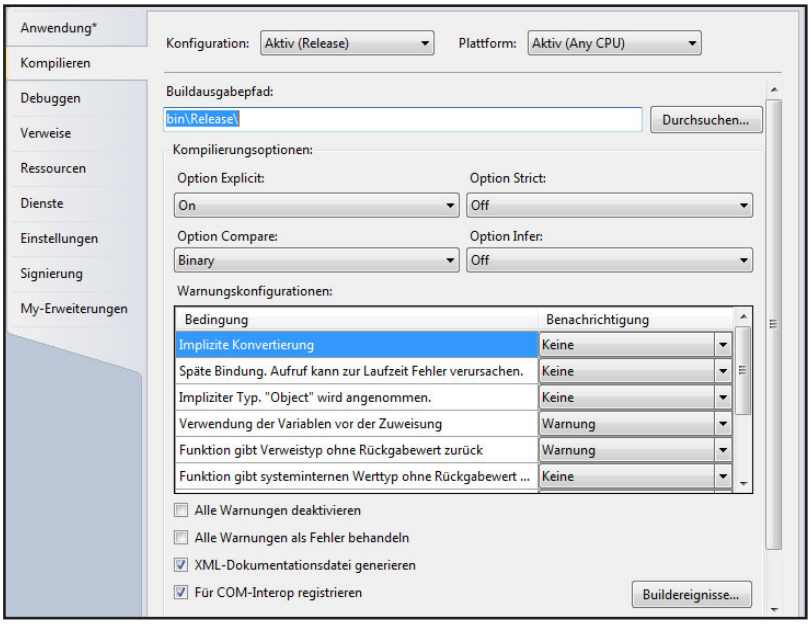


Abbildung 13.4
Für COM-Interop registrieren

Anschließend wollte ich natürlich die ganze Sache noch testen. Dazu habe ich ein VB 6-Projekt geöffnet und einen Verweis auf unsere Bibliothek gesetzt, die durch die automatische COM-Registrierung bereits in der Liste der verfügbaren Verweise vorhanden war, wie Sie in Abbildung 13.5 sehen können. Noch ein paar Zeilen Programmcode, um die Klasse zu instanziieren und aufzurufen, und schon war unsere .NET-Funktionalität in unserer VB 6-Anwendung verfügbar.

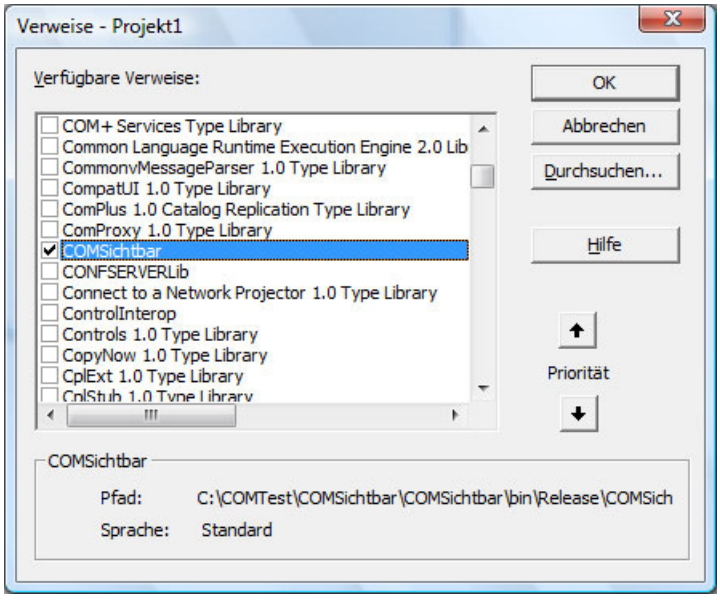


Abbildung 13.5
Verweis setzen aus einem VB6-Projekt

In Abbildung 13.6 sehen Sie das erfolgreiche Ergebnis und im Hintergrund auch noch den benötigten VB 6-Code.

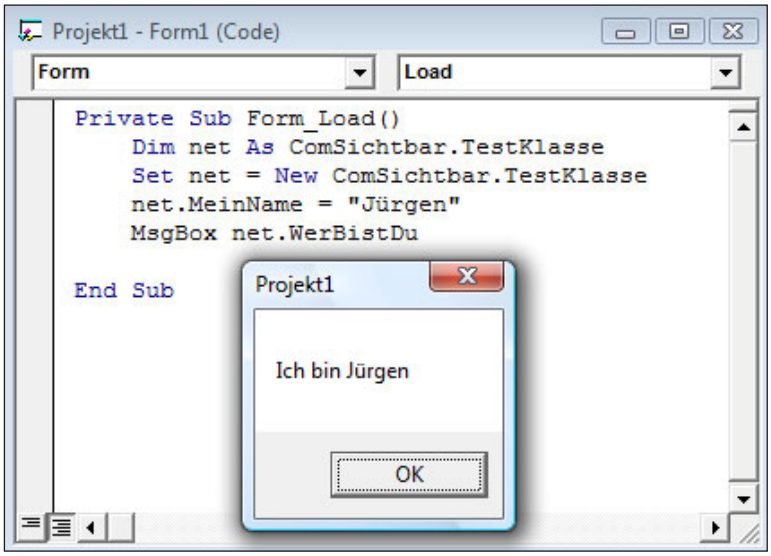


Abbildung 13.6
Aufruf der .NET Funktionalität aus VB6

14

ASP.NET 4.0

Im Verlauf dieses Buchs haben Sie die wichtigsten Sprachelemente von Visual Basic 10 im Einsatz gesehen. Der Fokus lag dabei auf der Desktop-Entwicklung (seien es Programme mit grafischer Benutzeroberfläche oder einfach nur Konsolenanwendungen). Einer der Hauptbestandteile von .NET ist jedoch noch ASP.NET. Dabei handelt es sich um den Nachfolger der »alten« Webtechnologie von Microsoft, ASP (Active Server Pages). Diese war einfach zu erlernen, aber dafür auch recht arm an Features. Mit .NET – und damit ASP.NET – hat sich das schlagartig geändert. Webanwendungen können nun auf die Features des .NET Framework zurückgreifen, was den mitgelieferten Funktionsumfang stark vervielfacht. Und mit dem .NET Framework 4.0 gibt es auch ASP.NET 4.0, das (unter anderem) das Ajax-Framework ASP.NET AJAX integriert hat.

Dieses Kapitel will einen ersten Einblick in ASP.NET 4.0 geben. Neben einigen allgemeinen Hintergründen werden die interessantesten Features (fraglos eine subjektive Auswahl) jeweils kurz anhand eines Beispiels vorgestellt. Der Fokus liegt dabei auf neuen Features von ASP.NET 2.0 bis 4.0, so dass auch Leser mit Vorkenntnissen in ASP.NET 1.0 oder 1.1 etwas davon haben. Kurze Werbepause: Im Buchangebot von Pearson Education finden Sie natürlich sowohl grundlegende als auch weiterführende Literatur zum Thema ASP.NET 4.0.

14.1 Installation und Hintergründe

Um ASP.NET (4.0 – die Versionsnummer wird im Folgenden weggelassen) zu verwenden, benötigen Sie eigentlich nur das .NET Framework. Allerdings wurde das Wort »eigentlich« bewusst gewählt: So richtig Sinn macht das Ganze nur, wenn Sie auch einen Webserver haben, auf dem dann ASP.NET-Seiten laufen. Bei Professional- und Server-Versionen von Windows, also etwa Windows XP Professional oder Windows Server 2003 oder Vista Business, ist ein Webserver dabei. Der hört auf den Namen IIS, das stand früher einmal für Internet Information Server und steht mittlerweile für Internet-Informationsdienste (Internet Information Services). Trotzdem verwendet man im üblichen Sprachgebrauch eher »der IIS« anstelle von »die IIS«.

Wenn Sie ein derartiges Windows-System haben, ist der Server unter Umständen noch nicht installiert, was Sie über Systemsteuerung/Software/Windows-Komponenten hinzufügen/entfernen (wie in Abbildung 14.1 zu sehen, heißt die Option bei den meisten Windows-Versionen Windows-Funktionen ein- oder ausschalten) nachholen können. Unter Windows 2003 und 2008 müssen Sie den Server zusätzlich so konfigurieren, dass er auch ASP.NET unterstützt (siehe Abbildung 14.2). Bei diesen Windows-Versionen ist das .NET Framework zwar automatisch dabei (allerdings in einer alten Version), aber beim Server ist aus Sicherheitsgründen standardmäßig vieles ausgeschaltet.

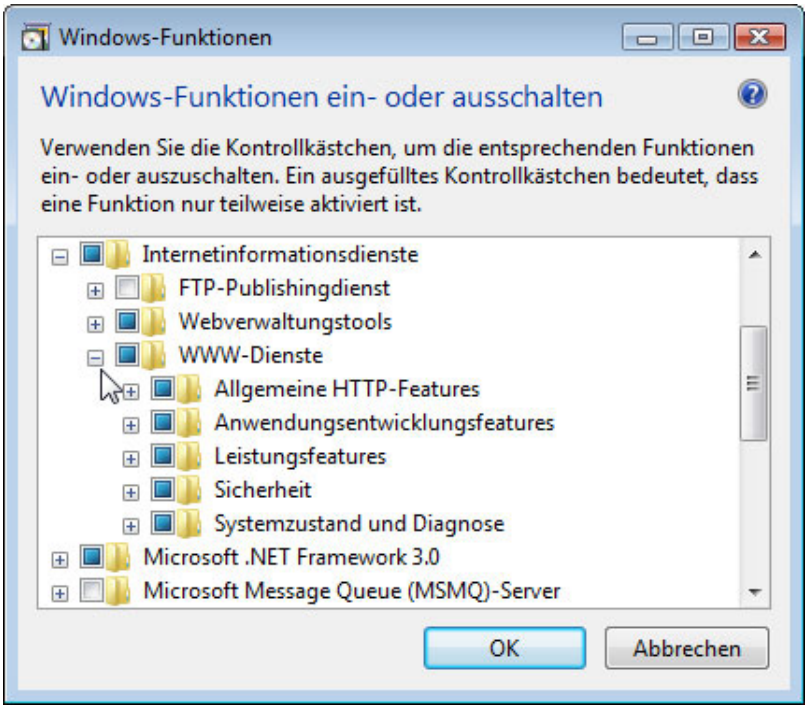


Abbildung 14.1
Installieren Sie den IIS als eine Windows-Komponente

Achtung

Gerade nach der Installation des IIS sollten Sie unverzüglich zu Windows Update gehen (z.B. Extras/Windows Update in der Menüleiste des Internet Explorers oder Kategorie System und Sicherheit/Windows Update in der Systemsteuerung), um eventuelle Sicherheitspatches für den Server einzuspielen. Ansonsten laufen Sie Gefahr, dass sich auf Ihrem System ein Virus oder andere Schadsoftware einnistet.

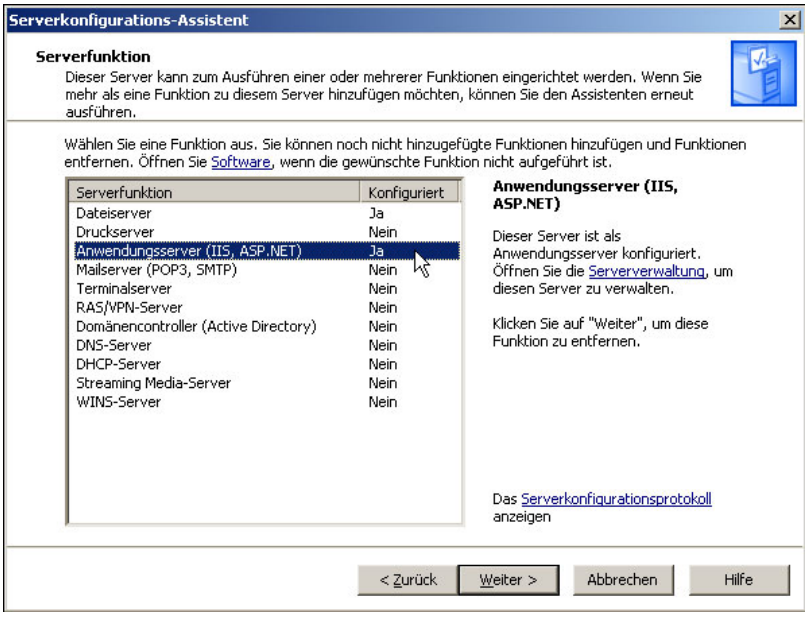


Abbildung 14.2
Unter Windows 2003 müssen Sie den Anwendungsserver extra für ASP.NET konfigurieren.

Achtung

ASP.NET ohne Microsoft

Es gibt natürlich auch noch andere Webserver (Weltmarktführer ist beispielsweise der Apache Webserver), aber diese unterstützen .NET nicht direkt oder nur über nicht offizielle Zusatzprojekte. Das bekannteste Projekt dieser Art ist Mono (*http://mono-project.com/*), das es sich zum Ziel gesetzt hat, eine Open-Source-Implementierung von .NET zu schaffen und das Framework damit auch auf anderen Betriebssystemen zum Laufen zu bekommen. Die Ergebnisse des Projekts sind erstaunlich; unter anderem gibt es eine Portierung von ASP.NET, die dann nicht nur unter Windows, sondern auch auf der Linux-Plattform und unter anderen Betriebssystemen läuft. Im Produktiveinsatz wird Mono äußerst selten eingesetzt und es schwingt immer ein wenig die Befürchtung mit, dass Microsoft irgendwann ein Patent oder Ähnliches aus der Schublade zieht, um das Projekt zu torpedieren; doch gerade wenn es darum geht, .NET auch auf anderen Plattformen zu verwenden, lohnt sich ein Blick.

Wird dann das .NET Framework installiert und ist ein Microsoft-Webserver vorhanden, integriert sich ASP.NET direkt in den Webserver. Alte .NET-Versionen (1.x) werden dabei allerdings nicht überschrieben, in der Management-Konsole des Webservers (START/AUSFÜHREN/INETMGR) können Sie das aber ändern. Beachten Sie, dass ASP.NET 4.0 das .NET Framework 4.0 als Zielversion verwendet. Wenn Sie die Website für 2.0 konfigurieren (siehe Abbildung 14.3), können Sie nur Features bis hin zur Version 3.5 verwenden.

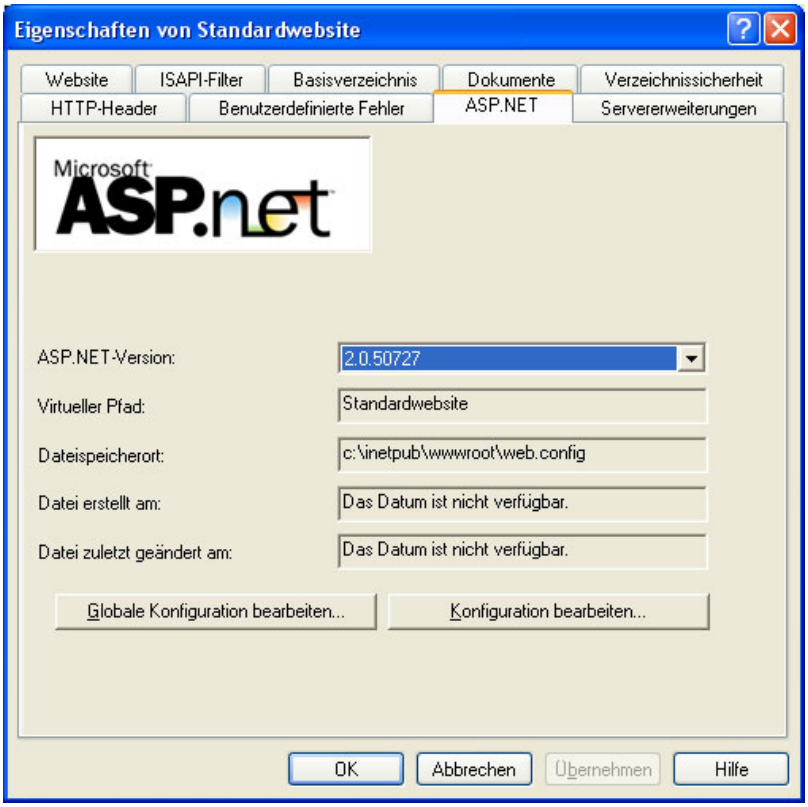


Abbildung 14.3
Stellen Sie ein, welche .NET-Version verwendet werden soll.

Zum Entwickeln benötigen Sie aber nicht einmal den IIS, Sie können sogar auf Windows XP Home setzen (das, so zumindest die Microsoft-Logik, als Privatanwender-Betriebssystem keinen Webserver benötigt). Denn: Von Microsoft gibt es einen wirklich guten Editor für ASP.NET-Webseiten. Der heißt »Visual Web Developer« (kurz: VWD) und ist in das Visual Studio 2010 mit integriert. Genauso wie Visual Basic 2010 Express gibt es auch den VWD als Express-Version, die kostenlos heruntergeladen werden kann. Unter *http://www.microsoft.com/germany/express/products/web.aspx* finden Sie einen Verweis auf den Web-Installer für das Tool – genauer gesagt auf den Microsoft Web Platform Installer, mit dem Sie auch weitere Produkte auf Ihrem System einrichten können.

Nach der Installation befindet sich in Ihrem Startmenü ein neuer, extrem breiter Eintrag: Microsoft Visual Studio 2010 Express. Ein Klick auf die enthaltene Verknüpfung Microsoft Visual Web Developer 2010 Express startet den Editor, der nicht nur optisch sehr an Visual Studio erinnert, sondern auch die Bedienung und viele Features damit gemein hat, wie Sie in Abbildung 14.5 sehen können.

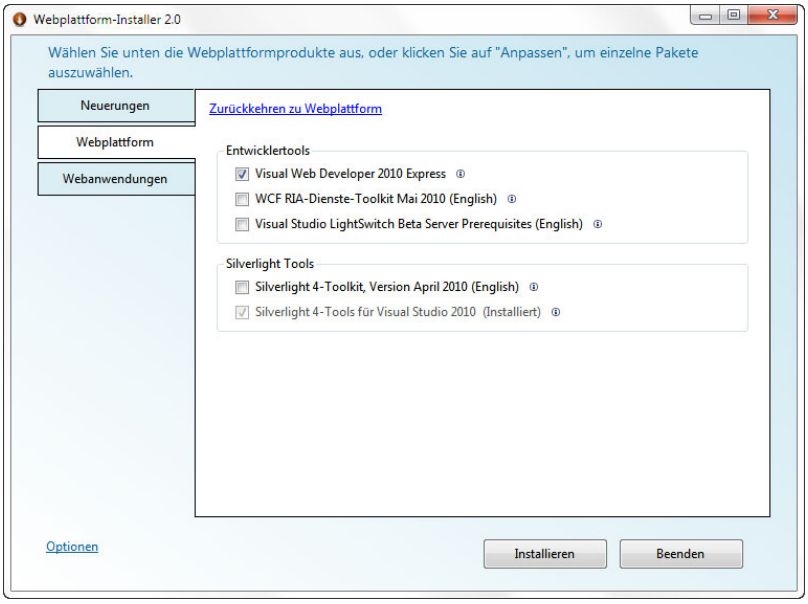


Abbildung 14.4
Die Installation erfolgt mit dem Web Platform Installer.

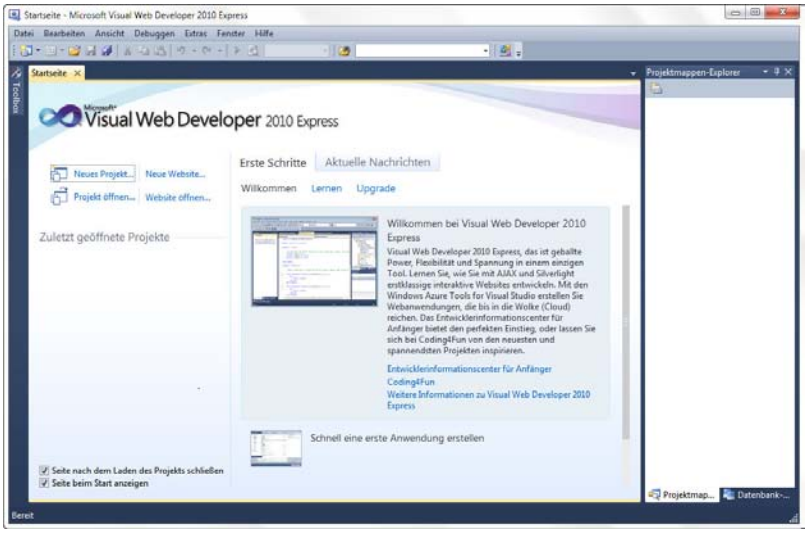


Abbildung 14.5
Der Startbildschirm des Visual Web Developer 2010 Express

Alle Beispiele in diesem Kapitel funktionieren natürlich ebenfalls mit der »Vollversion« von Visual Studio.

Info

Das Wichtigste in Kürze: Herkömmliche ASP.NET-Seiten haben die Dateiendung *.aspx* (es gibt zwar auch andere Dateiendungen, aber nur für spezielle Anwendungen wie etwa Web-Services). Das Ganze findet im Web und damit innerhalb einer Client-Server-Umgebung statt. Das bedeutet, dass eine ASP.NET-Seite auf dem Server ausgeführt wird, dem Client aber in der Regel nur HTML-Markup, CSS-Stilbefehle und JavaScript-Code liefert. In einer ASP.NET-Seite können Sie in jeder .NET-Sprache serverseitigen Code schreiben. Etwas ungewohnt ist allerdings das Ereignismodell – bereits gelernte VB-Tugenden aus der Windows Forms-Programmierung sind hierbei leider absolut ungeeignet. Sie können zwar schon auf clientseitige Ereignisse, also Geschehnisse im Webbrowser, reagieren, doch der Server kriegt davon erst dann etwas mit, wenn der Client auch tatsächlich Daten an ihn schickt.

Eine bequeme Möglichkeit, auf der Serverseite Daten zu hinterlegen, bietet die Methode `Page_Load()`. Sie wird beim Laden der Seite automatisch aufgerufen. Dort können Sie unter anderem Textausgaben erledigen. Dazu benötigen Sie die Methode `Write` des `Response`-Objekts – Sie schreiben also etwas in die Ausgabedaten.

Mit den folgenden Schritten erstellen Sie eine erste Testseite:

1. Erstellen Sie eine neue Website im Dateisystem (Datei/Neue Website); im Beispiel wird der Name *Kapitel14* verwendet. Achten Sie darauf, dass auch tatsächlich Visual Basic als Sprache ausgewählt wird, und selektieren Sie die Vorlage Leere ASP.NET-Website.

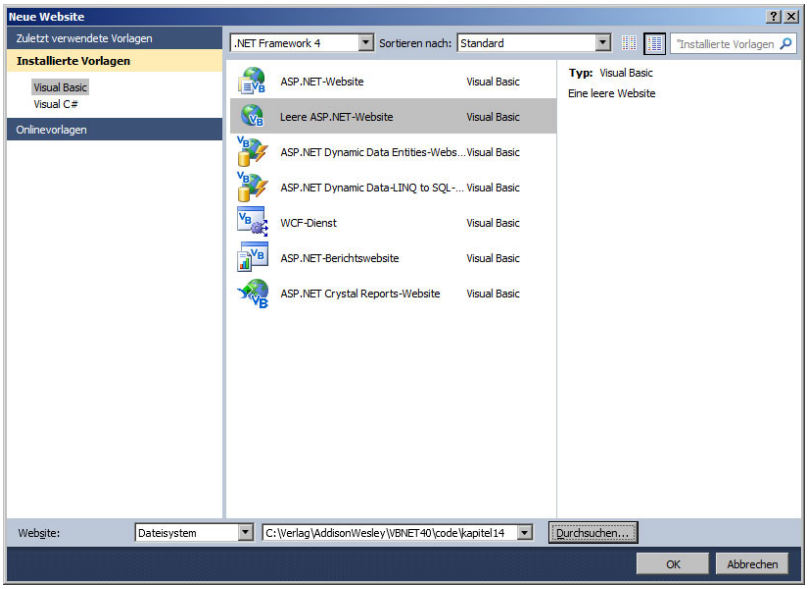


Abbildung 14.6
So legen Sie eine neue Website im Visual Web-Developer an.

2. Erzeugen Sie mit Datei/Neue Datei/Web Form eine Datei namens *Default.aspx* (wählen Sie auch hier Visual Basic als Sprache). Öffnen Sie diese per Doppelklick im Editor.
3. Wechseln Sie in die Entwurfsansicht und doppelklicken Sie auf die Oberfläche. Sie landen automatisch in der Code-Datei[1] für diese Seite, die den Namen *Default.aspx.vb* trägt.
4. Fügen Sie folgenden Code zwischen `Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load` und `End Sub` ein:

   ```
   Response.Write(DateTime.Now.ToShortTimeString())
   ```
5. Speichern Sie die Seite und laden Sie sie mit Strg+F5 im Webbrowser.

Sie sehen durch ein Informationsfeld im Systray, dass VWD einen lokalen Webserver startet, über den Sie die Seite anschauen können – bei einem Zugriff über das lokale Dateisystem, ohne Webserver, würden Sie nur den Quellcode der Seite zu Gesicht bekommen. So aber startet der Webserver ASP.NET und parst den Code in Ihrer *.aspx*-Datei und führt ihn zu guter Letzt aus. Das Ergebnis dieser Codeausführung landet dann beim Client. In Abbildung 14.7 sehen Sie die Ergebnisseite und den zugehörigen, von ASP.NET generierten Quellcode – übrigens mit ASP.NET 4.0 in perfektem XHTML! Die Visual Basic-Kommandos tauchen nicht auf, dafür wurde ganz am Anfang der Seite die aktuelle Uhrzeit ausgegeben.

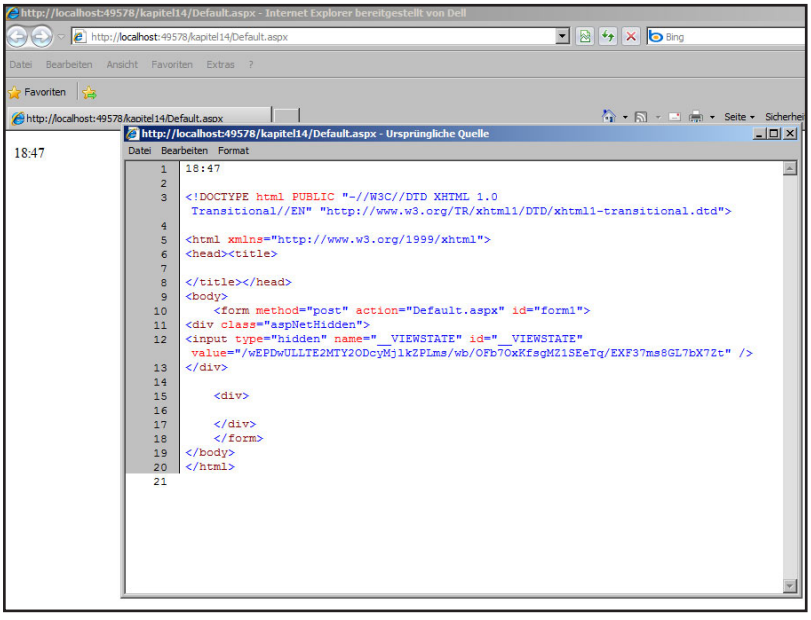


Abbildung 14.7
Die Webseite (hinten) und das dazugehörige HTML-Markup (vorne)

1 Das Prinzip heißt »Code behind«: Der eigentliche Code liegt in einer externen Datei. Somit sind Code und Inhalt voneinander getrennt, was aus Architektursicht sehr sinnvoll ist.

14.2 Steuerelemente

Das Beispiel ist bis dato ein wenig unbefriedigend, da die Daten am Anfang der HTML-Datei ausgegeben worden sind, Sie diese dynamischen Inhalte jedoch sicherlich gerne mitten auf der Seite platzieren möchten. Es gibt in ASP.NET eine Reihe von serverseitigen Steuerelementen, sogenannten Web Controls – vergleichbar mit den .NET-Steuerelementen der Windows Forms-Anwendungen. Viele Steuerelemente haben dabei sogar übereinstimmende Namen. Der große Clou: Die Daten in diesen Steuerelementen können Sie serverseitig steuern, also wie gehabt auf interessante Eigenschaften (etwa: `Text`) von Web Controls zugreifen.

ASP.NET unternimmt dabei große Anstrengungen, diese Ansteuerung recht intuitiv zu gestalten, wie man es etwa von Windows-Anwendungen gewohnt ist – sogar die Bezeichnungen der Steuerelemente sind in vielen Fällen identisch. Dass das nicht immer ohne Probleme ablaufen kann, ist offensichtlich, denn wie soll man serverseitig etwas setzen oder auslesen, das aber clientseitig abläuft? Das HTTP-Protokoll ist sehr einfach gestrickt: Verbindung aufbauen, Daten austauschen, Verbindung schließen. Sprich, sobald wieder mit dem Server kommuniziert werden muss, wird eine erneute HTTP-Verbindung benötigt. Es wird hier in ASP.NET recht viel herumgetrickst, mit JavaScript und anderen Techniken, so dass das halbwegs transparent funktioniert.

Hier ein kleines Beispiel, um das Prinzip zu verdeutlichen. In der Datei *Default.aspx* ziehen Sie in der Entwurfsansicht ein Label-Steuerelement auf die Oberfläche. Das erzeugt dann im Code folgendes Markup:

```
<asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>
```

Sie erkennen hier einige wesentliche Aspekte von ASP.NET:

- Web Controls beginnen mit `asp:`.
- Im `ID`-Attribut wird ein eindeutiger Bezeichner vergeben (wichtig später für den serverseitigen Zugriff!).
- Durch `runat="server"` wird ASP.NET mitgeteilt, dass es sich hierbei um ein serverseitiges Element (und nicht etwa um HTML, was nicht verändert werden darf) handelt.

Suchen Sie im Code der Datei *Default.aspx.vb* die Methode `Page_Load()` auf. Hier können Sie auf das Label-Steuerelement zugreifen. Nach Eingabe von `Label1.` aktiviert sich IntelliSense im Editor und bietet einen Zugriff auf die Objekteigenschaften und -methoden an. Schreiben Sie folgende Anweisung in die Methode `Page_Load()`:

```
Label1.Text = DateTime.Now.ToShortTimeString()
```

Damit wird also serverseitig dem Label-Steuerelement die aktuelle Uhrzeit als Text zugewiesen.

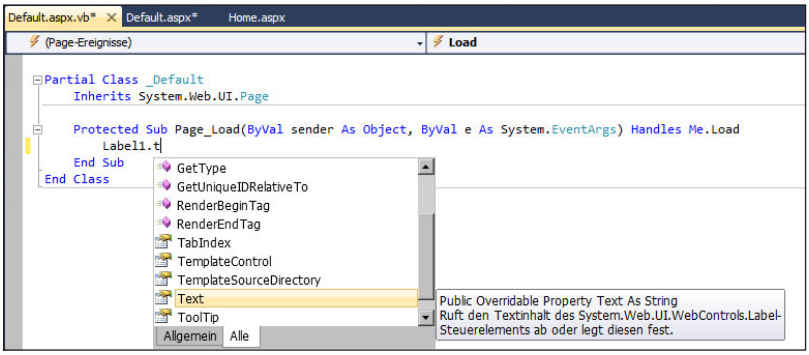


Abbildung 14.8
IntelliSense in Aktion

In Listing 14.1 sehen Sie den kompletten Code der Datei *Default.aspx.vb*:

Listing 14.1
Die Code-Behind-Datei (Default.aspx.vb)

```
Partial Class _Default
    Inherits System.Web.UI.Page

    Protected Sub Page_Load( _
        ByVal sender As Object, _
        ByVal e As System.EventArgs) _
        Handles Me.Load

        Label1.Text = _
            DateTime.Now.ToShortTimeString()
    End Sub
End Class
```

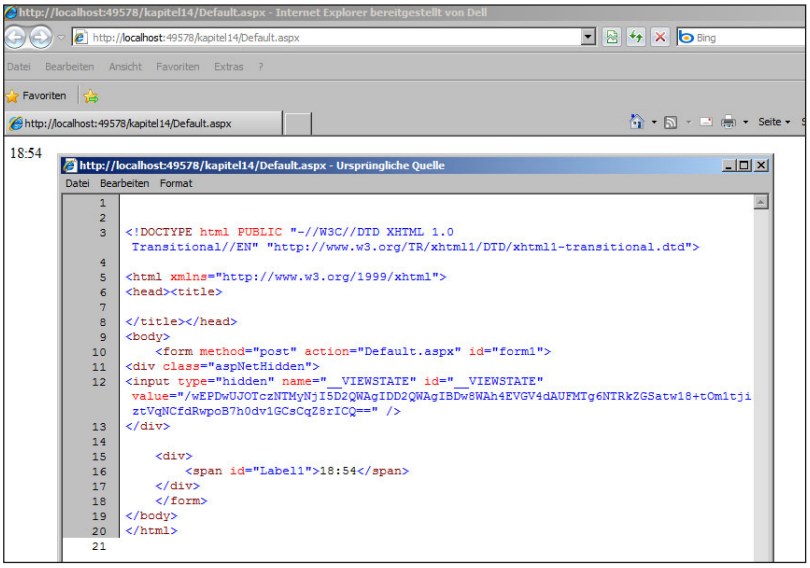
In der *Default.aspx* wird diese Datei per Direktive am Anfang der Seite geladen:

Listing 14.2
Die eigentlich leere ASP.NET-Datei – der Code lagert extern (Default.aspx).

```
<%@ Page Language="VB" AutoEventWireup="false" CodeFile="Default.
aspx.vb" Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Unbenannte Seite</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Label ID="Label1" runat="server" Text="Label"></
asp:Label></div>
  </form>
</body>
</html>
```

Führen Sie die Seite erneut im Webbrowser aus. Rein optisch ist die Ausgabe, die Sie in Abbildung 14.9 sehen, dieselbe wie zuvor, aber der Unterschied offenbart sich im Quellcode: Das Label-Element wurde dynamisch in das entsprechende HTML-Element (hier: `<span>`) umgesetzt.

Abbildung 14.9
Das Web Control wurde in HTML-Markup umgesetzt.



Achtung

Es gibt viele Web Controls, aber eines haben sie alle gemeinsam: Ohne `runat="server"` funktionieren sie nicht. In der WYSIWYG-Ansicht von Visual Web Developer wird dieses Attribut automatisch eingetragen, aber wenn Sie von Hand Markup in der Code-Ansicht eintragen, müssen Sie selbst dafür sorgen. Vergessen Sie es nicht, sonst liefert der Server an den Browser den Quellcode der Web Controls mit aus.

14.3 Masterseiten

Eine der Hauptneuerungen in ASP.NET 2.0 war ein integriertes Template-System. In der englischen Version heißt es Master Pages, die deutsche Übersetzung hat daraus Masterseiten gemacht. Das Prinzip ist sehr einfach: Es gibt einen neuen Dateityp mit der Endung *.master*. Dieser enthält sozusagen das Gerüst einer Website, das auf vielen Einzelseiten Verwendung findet: etwa die Kopf- und Fußzeile, die Navigation, den obligatorischen Impressumslink. Das ist eine Funktionalität, die viele Webeditoren wie etwa Macromedia Dreamweaver oder Microsoft Frontpage seit längerer Zeit anbieten, aber die Masterseiten von ASP.NET sind direkt in die Technologie integriert und somit auch performant und wartungsarm.

Um eine Masterseite anzulegen, wählen Sie den Menüpunkt Datei/Neue Datei und dort den Dateityp Masterseite. Der vorgegebene Dateiname, *MasterPage.master*, kann beibehalten werden. Die Checkbox Code in gesonderter Datei ablegen muss nicht angekreuzt sein, denn wir benötigen diesmal überhaupt keinen Code!

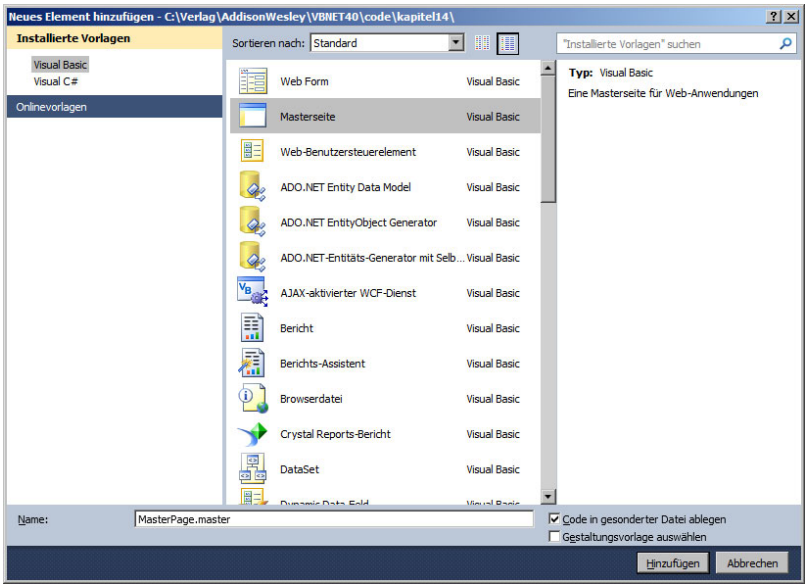


Abbildung 14.10
So legen Sie eine neue Masterseite an.

In dieser Masterseite können Sie dann ein Layout nach Wunsch anlegen (oder ein vorgefertigtes HTML-Markup eines Webdesigners einfügen). Wichtig ist, dass Sie mindestens einen Platzhalter vergeben. Das zugehörige Web Control heißt `ContentPlaceHolder` und befindet sich ebenfalls in der Toolbox am linken Rand des VWD. Die Masterseite enthält, wie gesagt, nur den Rumpf einer Webseite; alle Platzhalter werden dann auf den eigentlichen Inhaltsseiten mit Content gefüllt. Nachfolgend eine relativ einfach gehaltene Masterseite (die im Folgenden noch etwas erweitert wird) mit einem `ContentPlaceHolder`. Wichtig sind vor allem drei Dinge:

- Die Seite muss durch die Direktive `<%@ Master %>` am Dateianfang als Masterseite gekennzeichnet worden sein.
- Sie benötigen natürlich `runat="server"` bei dem `ContentPlaceHolder`.
- Jeder `ContentPlaceHolder` benötigt eine eigene, eindeutige ID.

Listing 14.3
Die Masterseite (MasterPage.master, Zwischenstand)

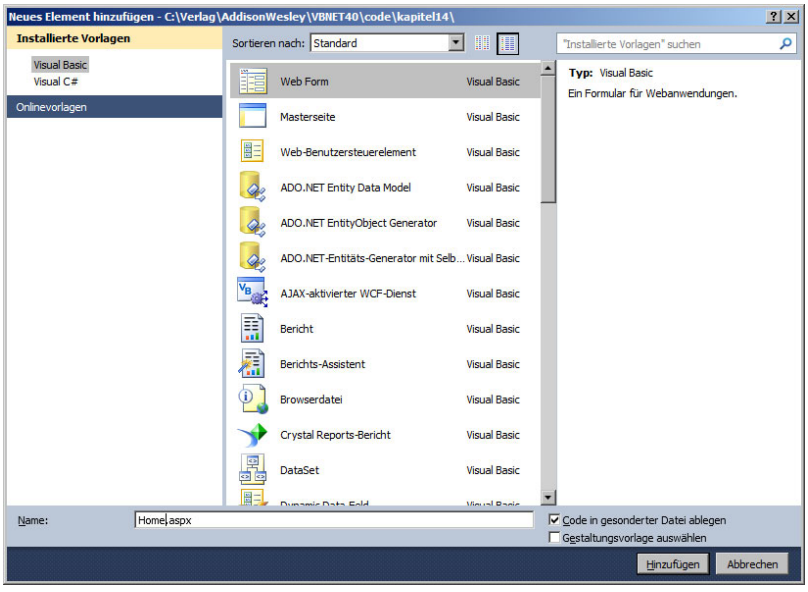
```
<%@ Master Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>ASP.NET</title>
</head>
<body>
  <table border="0" cellpadding="0" cellspacing="0" style="width:
   100%; height: 100%">
    <tr>
      <td align="center" style="height: 200px">
        <h1>
```

Listing 14.3 (Forts.) Die Masterseite (MasterPage.master, Zwischenstand)

```
          Willkommen!</h1>
        <br />
        <asp:HyperLink ID="HyperLink1" runat="server" Naviga-
         teUrl="Home.aspx">Homepage</asp:HyperLink>
        -
        <asp:HyperLink ID="HyperLink2" runat="server" Naviga-
         teUrl="Info.aspx">Informationen</asp:HyperLink>
        -
        <asp:HyperLink ID="HyperLink3" runat="server" Naviga-
         teUrl="Impressum.aspx">Impressum</asp:HyperLink></td>
    </tr>
    <tr>
      <td>
        <asp:ContentPlaceHolder ID="ContentPlaceHolder1"
         runat="server">
        </asp:ContentPlaceHolder>
      </td>
    </tr>
  </table>
</body>
</html>
```

Interessant wird es jetzt, wenn Sie eine neue ASP.NET-Seite auf Basis der Masterseite anlegen möchten. Dazu wählen Sie erneut DATEI/NEUE DATEI und als Dateityp WEB FORM. Klicken Sie jetzt die Checkbox GESTALTUNGSVORLAGE AUSWÄHLEN an. Daraufhin erscheint eine Liste aller in der Website angelegten Masterseiten. Das ist zwar zurzeit nur eine, aber immerhin: Wählen Sie sie aus und klicken Sie auf OK.

Abbildung 14.11 Geben Sie an, dass Sie eine Masterseite auswählen möchten!



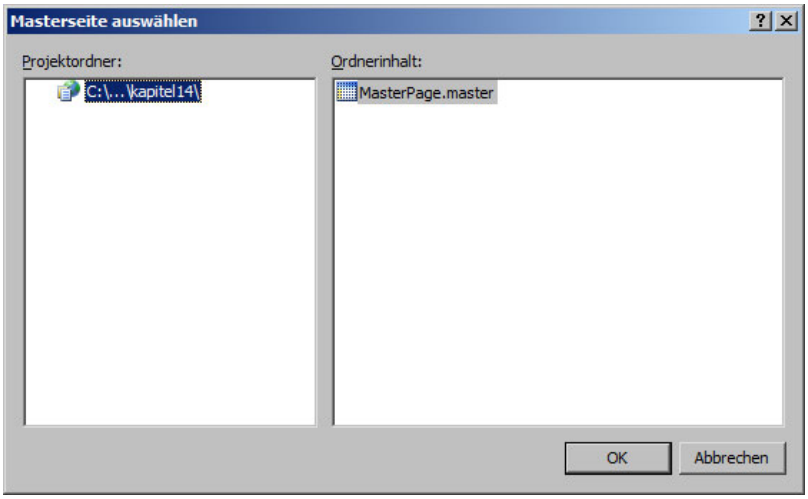


Abbildung 14.12
Daraufhin erscheint ein neues Dialogfeld.

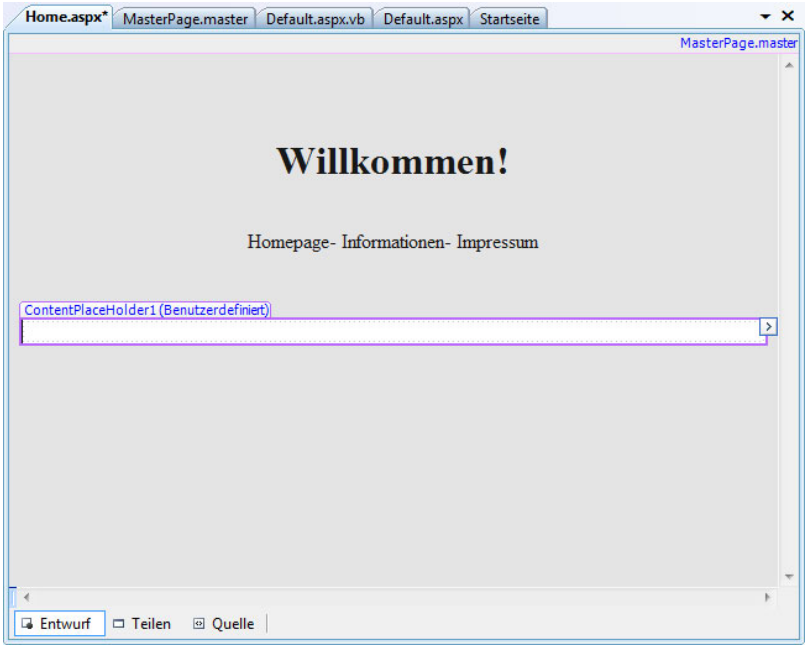
Im Editor wird dann eine relativ nackte Seite angelegt, die im Wesentlichen aus nur wenigen Markup-Elementen besteht:

```
<%@ Page Title="" Language="VB" MasterPageFile="~/MasterPage.
master" %>
<script runat="server">

</script>

<asp:Content ID="Content1" ContentPlaceHolde-
rID="ContentPlaceHolder1" Runat="Server">
</asp:Content>
```

In der WYSIWYG-Ansicht sehen Sie auch die im Code per Eigenschaft `MasterPageFile` referenzierte Masterseite. Sie ist zwar ausgegraut, jedoch lässt sich erahnen, wie sich das Ganze später im Webbrowser darstellen wird.

Das weitere Vorgehen ist dann ganz einfach. Für jeden `ContentPlaceHolder` hat Visual Web Developer ein `Content`-Web-Control angelegt. Welcher `Content` zu welchem `ContentPlaceHolder` gehört, wird über das Attribut `ContentPlaceHolderID` angegeben – jetzt sehen Sie auch, wieso jeder `ContentPlaceHolder` eine eigene ID benötigt. Innerhalb von `<asp:Content>`, natürlich wieder mit `runat="server"`, geben Sie den Inhalt an, der an die Stelle des Platzhalters rücken soll. In Listing 14.4 sehen Sie ein Beispiel für die Datei *Home.aspx*; erstellen Sie nach derselben Machart die weiteren Seiten *Info.aspx* und *Impressum.aspx*.

Abbildung 14.13
Die von der Masterseite abgeleitete Einzelseite

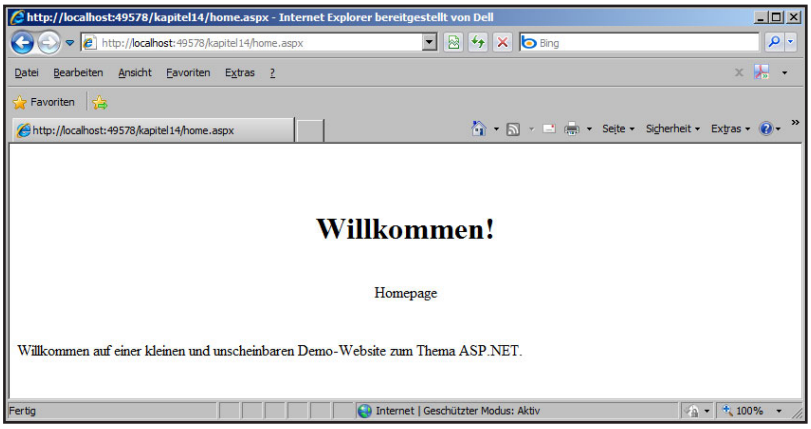


Listing 14.4
Die von der Masterseite abgeleitete ASP.NET-Seite (Home.aspx)

```
<%@ Page Language="VB" MasterPageFile="~/MasterPage.master" %>
<asp:Content ID="Content1" ContentPlaceHolderID=
"ContentPlaceHolder1" Runat="Server">
Willkommen auf einer kleinen und unscheinbaren Demo-Website zum
Thema ASP.NET.
</asp:Content>
```

Der abschließende Blick gilt natürlich wie immer dem Webbrowser. Und in der Tat: ASP.NET setzt die Masterseite und die Inhaltsseite zusammen und erstellt daraus eine einzelne Seite. Ein Endbenutzer bekommt so gar nicht mit, dass hier überhaupt ein Templating-System zum Einsatz gekommen ist.

Abbildung 14.14
Die zusammengesetzte Seite im Webbrowser



14.4 Navigation

Wie Sie im vorherigen Beispiel gesehen haben, ist die Navigation für die Website in der Master Page untergebracht, jedoch statisch eingefügt. Wenn sie auf einer Unterseite etwas anders aussehen soll (beispielsweise falls der aktuell ausgewählte Navigationspunkt nicht mehr verlinkt sein soll), müssen Sie grundsätzlich programmieren. Allerdings liegt einer der Schwerpunkte von ASP.NET 4.0 (und auch von ASP.NET 2.0, der Version, in der dieses Feature eingeführt worden ist) auf der Reduzierung von Code, so dass sich der Programmieraufwand auf ein Mindestmaß beschränken kann. Für die Navigation liefert ASP.NET 4.0 drei verschiedene, spezielle Web Controls mit, die hierzu diverse Aspekte implementieren.

Zunächst benötigen Sie eine Sitemap. Das ist eine XML-Datei, die strukturelle Informationen über den Aufbau Ihrer Website zur Verfügung stellt. Im Visual Web Developer legen Sie mit DATEI/NEUE DATEI und dem Dateityp SITE-ÜBERSICHT (es lebe die krampfhafte Eindeutschung feststehender Begriffe) die Datei *Web.sitemap* an. Sie erhalten schon ein Grundgerüst des benötigten XML-Markups, müssen das aber noch mit Inhalten füllen und erweitern. Grundsätzlich ist festzuhalten:

- Der Wurzelknoten der Siteübersicht ist `<siteMap>`.
- Jeder einzelne Knoten der Sitemap wird mit `<siteMapNode>` dargestellt und bezeichnet eine Datei auf der Website.
- Durch Verschachteln von Knoten können Sie Hierarchien angeben.
- Innerhalb von `<siteMapNode>` geben Sie eine URL (Eigenschaft `url`), einen Titel (Eigenschaft `title`) und eine Kurzbeschreibung (Eigenschaft `description`) für die jeweilige Unterseite an.

Für unsere Minianwendung (Homepage, Info-Seite, Impressum) ist folgende Site-Übersicht ein ganz guter Anfang:

Listing 14.5
Die Site-Übersicht der Mini-Website (Web.sitemap)

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-
File-1.0" >
    <siteMapNode url="Home.aspx" title="Homepage" descrip-
     tion="Startseite der Anwendung">
        <siteMapNode url="Info.aspx" title="Informationen" de-
         scription="Allgemeine Informationen über die Website" />
        <siteMapNode url="Impressum.aspx" title="Impressum" de-
         scription="Impressum und Kontakt" />
    </siteMapNode>
</siteMap>
```

Sie haben jetzt die Struktur der Seite in XML abgebildet, aber können damit noch nichts anfangen. Die Betonung liegt auf noch, denn im nächsten Schritt verwenden Sie Ihre Site-Übersicht. Laden Sie dazu erneut die Master Page und löschen Sie alle Links auf die Unterseiten, denn das geschieht jetzt automatisch per Sitemap. In der Toolbox finden Sie in der Rubrik NAVIGATION das Element *SiteMapPath*. Fügen Sie es per Drag&Drop ein, wodurch folgendes Markup in der Site platziert wird:

```
<asp:SiteMapPath ID="SiteMapPath1" runat="server">
</asp:SiteMapPath>
```

Und das war es auch schon! Die XML-Daten werden automatisch eingelesen und daraus eine Navigation erstellt (die natürlich flexibel anpassbar ist). Beachten Sie den Unterschied zwischen Abbildung 14.15 und Abbildung 14.16: Auf der Homepage sehen Sie nur, wo Sie sind; auf Unterseiten sehen Sie den kompletten hierarchischen Navigationspfad bis zur Zielseite. Das nennt man übrigens Breadcrumb-Navigation, in Anlehnung an die (letztendlich nicht ganz optimale) Zielführung im Märchen Hänsel und Gretel.

Abbildung 14.15
Auf der Homepage erscheint nur die aktuelle Seite.

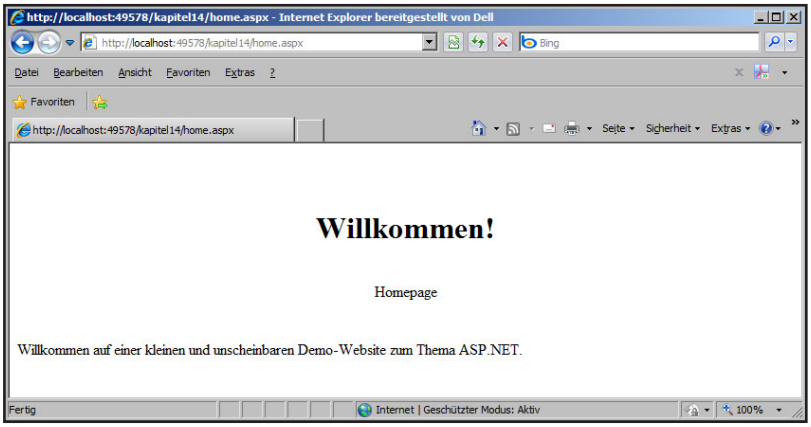
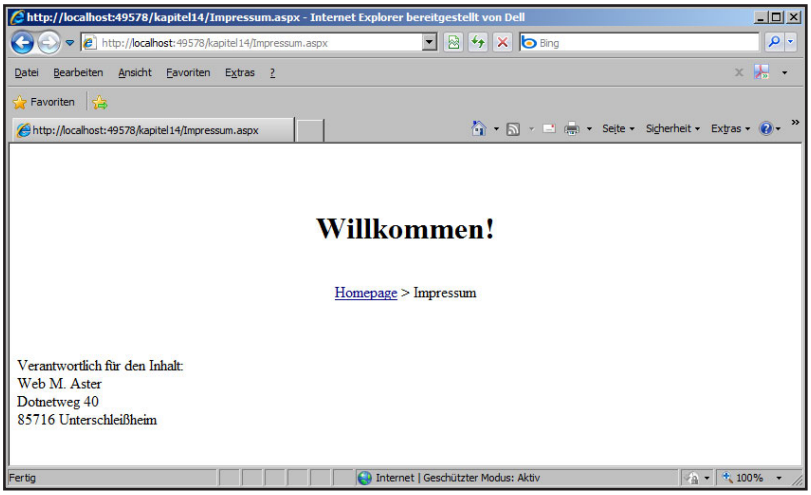


Abbildung 14.16
Auf Unterseiten können Sie die Struktur sehen.



Doch damit kommen Sie immer noch nicht auf alle wichtigen Seiten Ihrer Website. Dazu dient ein weiteres Navigationselement von ASP.NET 4.0, das Web Control `Menu`. Auch dieses fügen Sie in der WYSIWYG-Ansicht auf die Seite ein. Im Smart-Tag wählen Sie zunächst die Datenquelle. Da Sie noch

keine derartige haben, müssen Sie eine neue anlegen. Sie ahnen möglicherweise bereits, worauf das hinausläuft: Sie nehmen einfach die Sitemap.

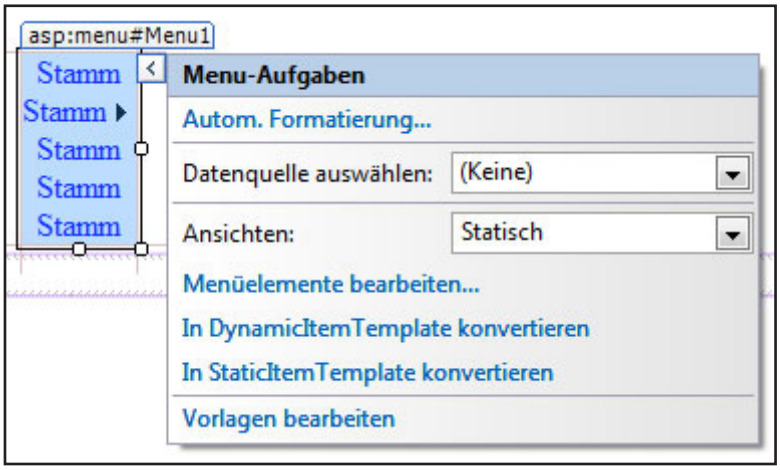


Abbildung 14.17
Legen Sie eine neue Datenquelle an.

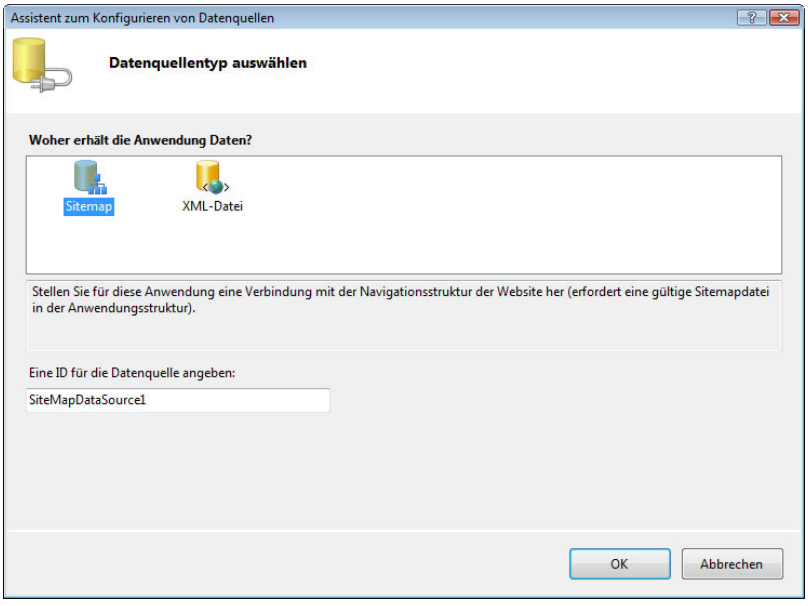


Abbildung 14.18
Am bequemsten ist es natürlich, bei der Datenquelle auf die Sitemap zu setzen.

Wieder wurde nur minimaler Markup-Code in die Seite eingefügt:

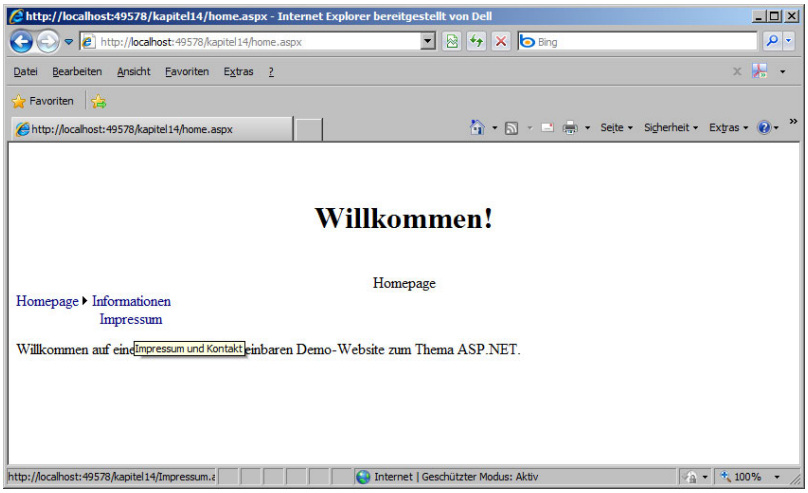
```
<asp:Menu ID="Menu1" runat="server" DataSour-
ceID="SiteMapDataSource1">
</asp:Menu>
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
```

Eventuell müssen Sie jetzt noch ein kleines Versäumnis nachholen: Das Ganze funktioniert nur, wenn sich `<asp:Menu>` innerhalb eines ASP.NET-Formulars befindet. Dieses Formular muss natürlich ebenfalls `runat="server"` verwenden:

```
<form runat="server">
<asp:Menu ID="Menu1" runat="server" DataSour-
ceID="SiteMapDataSource1" Orientation="Horizontal">
</asp:Menu>
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
</form>
```

Und schon haben Sie, wie Abbildung 14.19 für den Internet Explorer zeigt, eine automatisch generierte, dynamische JavaScript-Navigation, die natürlich auch in anderen Browsern funktioniert.

Abbildung 14.19
Die JavaScript-Navigation



Hier noch einmal der komplette Code für die Masterseite inklusive Navigationselementen:

Listing 14.6
Die Masterseite inklusive Navigation (MasterPage.master)

```
<%@ Master Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>ASP.NET</title>
</head>
<body>
  <form runat="server">
  <table border="0" cellpadding="0" cellspacing="0" style="width:
   100%; height: 100%">
```

Listing 14.6 (Forts.) Die Masterseite inklusive Navigation (MasterPage.master)

```
    <tr>
      <td align="center" style="height: 200px">
        <h1>
          Willkommen!</h1>
        <asp:SiteMapPath ID="SiteMapPath1" runat="server">
        </asp:SiteMapPath>
        <br />
        <asp:Menu ID="Menu1" runat="server" DataSourceID=
         "SiteMapDataSource1">
        </asp:Menu>
        <asp:SiteMapDataSource ID="SiteMapDataSource1"
         runat="server" />
        </td>
    </tr>
    <tr>
      <td>
        <asp:ContentPlaceHolder ID="ContentPlaceHolder1"
         runat="server">
        </asp:ContentPlaceHolder>
      </td>
    </tr>
  </table>
  </form>
</body>
</html>
```

Info

Baumansicht

Wie bereits angesprochen, gibt es noch ein drittes Navigationselement. Das heißt `TreeView` und implementiert eine Baumansicht, wie sie beispielsweise vom Windows Explorer her bekannt ist. Das Ganze gab es bereits für ASP.NET 1.x in Form von ASP.NET IE Web Controls, jedoch stark für den Namensgeber Internet Explorer optimiert (sprich: in anderen Browsern kein JavaScript-Komfort). Die `TreeView`-Variante von ASP.NET 4.0 ist browser-agnostisch und somit mehr oder minder universell einsetzbar. Auch hier benötigen Sie wieder eine Datenquelle wie etwa die XML-Sitemap. Außerdem gibt es eine Reihe von Anpassungsmöglichkeiten und per Schnittstelle (API) können Sie das Ganze auch programmieren.

Abschließend werfen wir noch einen Blick auf ein Web Control, das mit dem Thema Navigation recht eng verwandt ist: einen Assistenten. Diese kennen Sie aus der Windows-Programmierung, im Web sind sie etwas schwieriger zu implementieren. Aufgrund der Eigenheiten des HTTP-Protokolls müssen Sie bei jedem Schritt des Assistenten alle Daten irgendwo persistieren, um sie beim Vor- oder Zurückspringen parat zu haben.

Bei ASP.NET 4.0 gibt es eine Lösung in Form des `<asp:Wizard>`-Web-Controls. Innerhalb des Controls können Sie im Abschnitt `<WizardSteps>` die einzelnen Schritte des Assistenten angeben. Jeder Schritt wird durch das Web Control `<asp:WizardStep>` gekennzeichnet. Per Event-Handling können Sie

auf das Springen zwischen den einzelnen Schritten zugreifen. Erstellen Sie also eine neue Seite auf Basis der Masterseite – diesmal wieder inklusive Code-Behind-Datei – und legen Sie dort den folgenden Assistenten mit drei Schritten an:

Listing 14.7 Der Assistent (Wizard.aspx)

```
<%@ Page Language="VB" MasterPageFile="~/MasterPage.master"
AutoEventWireup="false"
    CodeFile="Wizard.aspx.vb" Inherits="Wizard" Title="ASP.NET" %>

<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlace
 Holder1" runat="Server">
    <div>
        <asp:Wizard ID="Wizard1" runat="server" ActiveSte-
         Index="0" CellPadding="10">
            <WizardSteps>
                <asp:WizardStep Title="Schritt 1" ID="StepName"
                 runat="server">
                    Ihr Name:
                    <asp:TextBox ID="tbName" runat="server" />
                </asp:WizardStep>
                <asp:WizardStep Title="Schritt 2" ID="StepMail"
                 runat="server">
                    Ihre E-Mail-Adresse:
                    <asp:TextBox ID="tbMail" runat="server" />
                </asp:WizardStep>
                <asp:WizardStep Title="Schritt 3" ID="StepFertig"
                 runat="server">
                    <p>
                        Vielen Dank für Ihre Angaben!</p>
                    <p>
                        Name:
                        <asp:Label ID="lblName" runat="server"
/><br />
                        E-Mail:
                        <asp:Label ID="lblMail" runat="server"
/></p>
                </asp:WizardStep>
            </WizardSteps>
        </asp:Wizard>
    </div>
    <div>
        Seite generiert am/um
        <asp:Label ID="lblZeit" runat="server" />
    </div>
</asp:Content>
```

Beim Springen im Assistenten (Seite vor, Seite zurück) tritt das Ereignis `ActiveStepChanged` auf. Dies lässt sich in der Code-Behind-Datei abfangen. Ist nämlich Schritt drei erreicht, sollen die Eingaben aus den Textfeldern wieder ausgegeben werden. Die Eigenschaft `ActiveStepIndex` des Assistenten liefert die Nummer des aktuellen Schritts, wobei die Zählung bei 0 beginnt – der dritte Schritt hat also den Wert 2. Damit lässt sich der Code wie folgt schreiben:

```
Protected Sub Wizard1_ActiveStepChanged( _
    ByVal sender As Object, _
    ByVal e As System.EventArgs) _
    Handles Wizard1.ActiveStepChanged

    If Wizard1.ActiveStepIndex = 2 Then
        lblName.Text = _
            HttpUtility.HtmlEncode(tbName.Text)
        lblMail.Text = _
            HttpUtility.HtmlEncode(tbMail.Text)
    End If
End Sub
```

Info

Die Methode `HttpUtility.HtmlEncode()` befreit die Eingabe von etwaigen Sonderzeichen wie etwa spitzen Klammern.

Außerdem befindet sich in der Code-Behind-Datei noch die Methode `Page_Load()`, in der bei jedem Neuladen der Seite die aktuelle Zeit ausgegeben wird – der Sinn dieses Vorgehens offenbart sich an späterer Stelle in diesem Kapitel. Hier zunächst der komplette Code der Code-Behind-Datei:

Listing 14.8
Die Code-Behind-Datei für den Assistenten (Wizard.aspx.vb)

```
Partial Class Wizard
    Inherits System.Web.UI.Page

    Protected Sub Wizard1_ActiveStepChanged( _
        ByVal sender As Object, _
        ByVal e As System.EventArgs) _
        Handles Wizard1.ActiveStepChanged

        If Wizard1.ActiveStepIndex = 2 Then
            lblName.Text = _
              HttpUtility.HtmlEncode(tbName.Text)
            lblMail.Text = _
              HttpUtility.HtmlEncode(tbMail.Text)
        End If
    End Sub

    Protected Sub Page_Load( _
        ByVal sender As Object, _
        ByVal e As System.EventArgs) _
        Handles Me.Load

        lblZeit.Text = _
            DateTime.Now.ToLongTimeString()
    End Sub

End Class
```

Sie können nun im Wizard navigieren und sehen im dritten Schritt Ihre vorherigen Eingaben wieder.

Abbildung 14.20
Schritt 1 des Assistenten

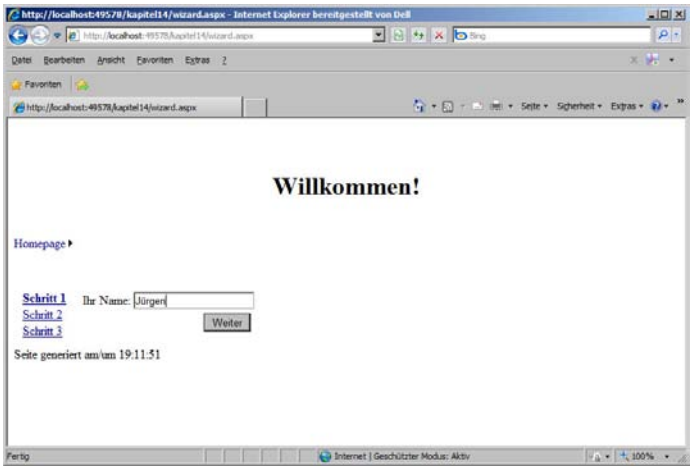


Abbildung 14.21
Schritt 2 des Assistenten

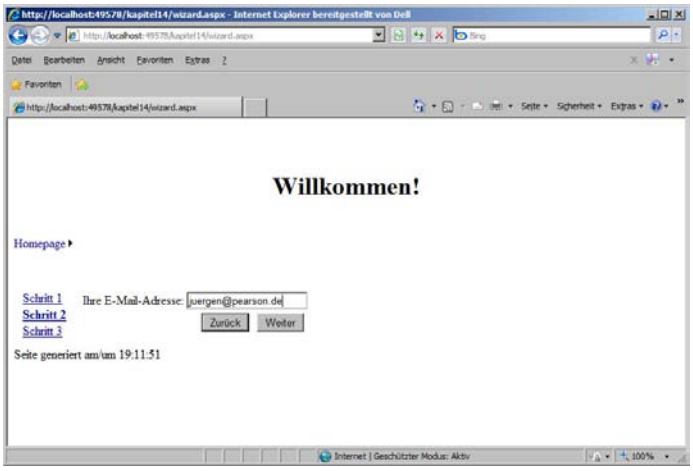
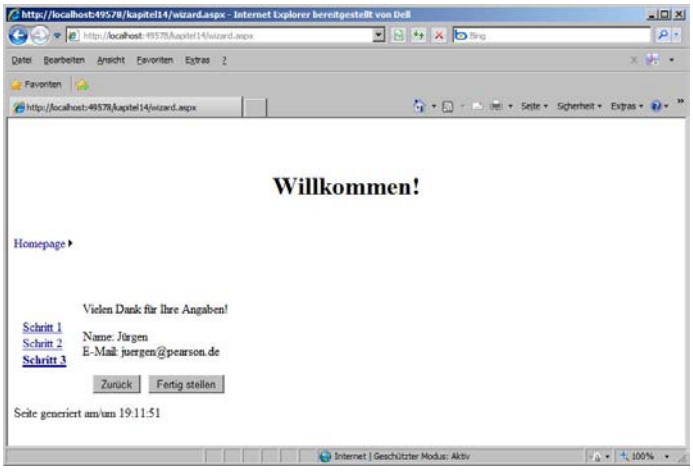


Abbildung 14.22
Schritt 3 des Assistenten



14.5 Ajax

Im Februar 2005 hat ein kalifornischer Usability-Experte namens Jesse James Garrett einen bahnbrechenden Artikel namens »Ajax: A New Approach to Web Applications« geschrieben (online unter *http://adaptivepath.com/publications/essays/archives/000385.php*). Dort beschreibt er ein technisches Konzept, um bei Websites den gefürchteten Page-Refresh zu vermeiden – das komplette Neuladen einer Seite, obwohl sich nur kleine Details verändern. Die dahinterliegende Technik – hauptsächlich ein geschickter Einsatz der clientseitigen Programmiersprache JavaScript – war damals keineswegs neu, aber Garretts Verdienst liegt darin, dass er das Konzept propagiert und mit einem schmissigen Namen versehen hat.

Microsoft ist auf den damals schnell fahrenden Ajax-Zug aufgesprungen und hat an einem Ajax-Framework gearbeitet, das sich möglichst nahtlos in ASP.NET integrieren sollte. Der Codename des Frameworks, das erstmals im September 2005 vorgestellt wurde, war Atlas. Die Ende Januar 2007 erschienene, finale Version hieß dann ASP.NET AJAX.

Das Ajax-Framework war so ein Erfolg, dass sich Microsoft entschlossen hat, es ab ASP.NET 3.5 – und damit auch in ASP.NET 4.0 – fest zu integrieren. ASP.NET AJAX ist also in der neuen .NET-Version automatisch mit dabei. Es ist kein Zusatz-Download oder eine Extrakonfiguration notwendig. Wenn Sie wie oben eine neue Website in Visual Studio oder Visual Web Developer erstellen, ist sie bereits komplett für Ajax vorkonfiguriert.

Im Folgenden greifen wir einen bestimmten Aspekt von ASP.NET AJAX heraus: Wir möchten auf das komplette Neuladen der Seite verzichten, wenn ein weiterer Schritt im Assistenten angesteuert wird. Wenn Sie einen Blick auf die Uhrzeiten in Abbildung 14.20 bis Abbildung 14.22 werfen, sehen Sie, dass diese jeweils unterschiedlich sind. Die komplette Seite wurde also neu geladen.

Um dies zu vermeiden, sind lediglich zwei Schritte zu tun. Zunächst müssen Sie ein neues Web Control in die Seite einfügen: `<asp:ScriptManager>`. Vergeben Sie eine ID und das bekannte Attribut `runat="server"`.

```
<asp:ScriptManager id="ScriptManager1" runat="server" />
```

Der nächste Schritt führt ein weiteres neues Web Control ein: `<asp:UpdatePanel>`. Innerhalb des UpdatePanel gibt es den Abschnitt `<ContentTemplate>`. In diesen verschieben Sie das komplette Markup des `<asp:Wizard>`-Steuerelements. Sie sollten dann bei folgendem Code angekommen sein:

Listing 14.9
Der mit Ajax angereicherte Assistent (Wizard.aspx)

```
<%@ Page Language="VB" MasterPageFile="~/MasterPage.master"
AutoEventWireup="false"
    CodeFile="Wizard.aspx.vb" Inherits="Wizard" Title="ASP.NET" %>
```

Listing 14.9 (Forts.) Der mit Ajax angereicherte Assistent (Wizard.aspx)

```
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlace
 Holder1" runat="Server">
    <div>
      <asp:ScriptManager ID="ScriptManager1" runat="server" />
      <asp:UpdatePanel ID="UpdatePanel1" runat="server">
        <ContentTemplate>
        <asp:Wizard ID="Wizard1" runat="server" ActiveStep
         Index="0" CellPadding="10">
            <WizardSteps>
                <asp:WizardStep Title="Schritt 1" ID="StepName"
                 runat="server">
                    Ihr Name:
                    <asp:TextBox ID="tbName" runat="server" />
                </asp:WizardStep>
                <asp:WizardStep Title="Schritt 2" ID="StepMail"
                 runat="server">
                    Ihre E-Mail-Adresse:
                    <asp:TextBox ID="tbMail" runat="server" />
                </asp:WizardStep>
                <asp:WizardStep Title="Schritt 3" ID="StepFertig"
                 runat="server">
                    <p>
                        Vielen Dank für Ihre Angaben!</p>
                    <p>
                        Name:
                        <asp:Label ID="lblName" runat="server"
/><br />
                        E-Mail:
                        <asp:Label ID="lblMail" runat="server"
/></p>
                </asp:WizardStep>
            </WizardSteps>
        </asp:Wizard>
        </ContentTemplate>
      </asp:UpdatePanel>
    </div>
    <div>
        Seite generiert am/um
        <asp:Label ID="lblZeit" runat="server" />
    </div>
</asp:Content>
```

Und das war es auch schon! Wenn Sie das Beispiel im Browser ausführen und im Wizard die einzelnen Schritte aufsuchen, so funktioniert dieser weiterhin wie vorher – aber die Uhrzeit bleibt dieselbe, es findet also kein komplettes Neuladen der Seite statt. Mit einem Browser-Plug-in wie etwa Firebug für Firefox (*http://www.getfirebug.com/*) oder dem Web Dev Helper für den Internet Explorer (*http://projects.nikhilk.net/Projects/WebDevHelper.aspx*) sehen Sie, dass aber tatsächlich mit dem Webserver Daten ausgetauscht werden – JavaScript und Ajax sei Dank.

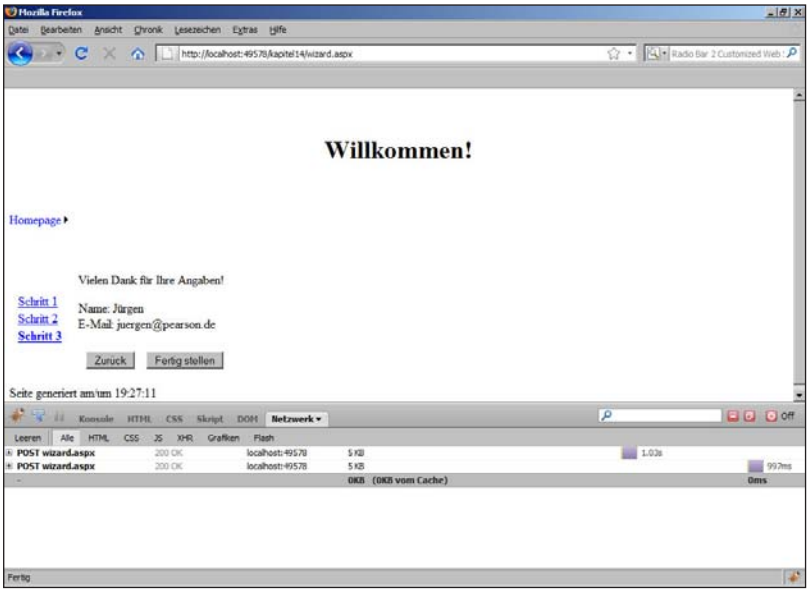


Abbildung 14.23
Der Assistent funktioniert weiterhin, die Uhrzeit bleibt unverändert.

14.6 Ausblick

Dieses Kapitel konnte natürlich nur einen ersten Einblick in ASP.NET 4.0 bieten. Die Technologie selbst bietet viele weitere praktische, nützliche, aber auch professionelle Möglichkeiten, eine dynamische und moderne Website zu erstellen. Zu den Features gehören unter anderem:

- Spezielle Web Controls, die eine Zwei-Wege-Anbindung an Datenbanken bieten
- Die meisten Web Controls sind stark anpassbar, so dass ein Layout auch für vorgefertigte Komponenten übernommen werden kann.
- Programmierschnittstellen (APIs) bieten einen einfachen Zugriff auf die Prozesse im Hintergrund; so können beispielsweise auch eigene Sitemap-Provider implementiert werden.
- Benutzerverwaltung und Rollenmanagement sind integriert, so dass sich geschützte Bereiche ohne großen Aufwand erstellen lassen.
- Viele weitere Web Controls erledigen Aufgaben, die man sonst in der Regel »zu Fuß« erledigt hat; das verspricht einen großen Produktivitätsgewinn.
- Das gesamte .NET Framework steht zur Verfügung; damit kann etwa auch eine Geschäftslogik einer Desktop-.NET-Anwendung (in Form einer Assembly) verwendet werden.

Wenn Sie also Websites erstellen möchten (oder müssen), ist ASP.NET sicherlich eine gute Wahl – und die Witze, die noch über den kaum vorhandenen Funktionsumfang des klassischen ASP gerissen werden konnten, sind bei ASP.NET ohne Pointe.

Stichwortverzeichnis

D

E

F

N

O

P

R

S

T

U

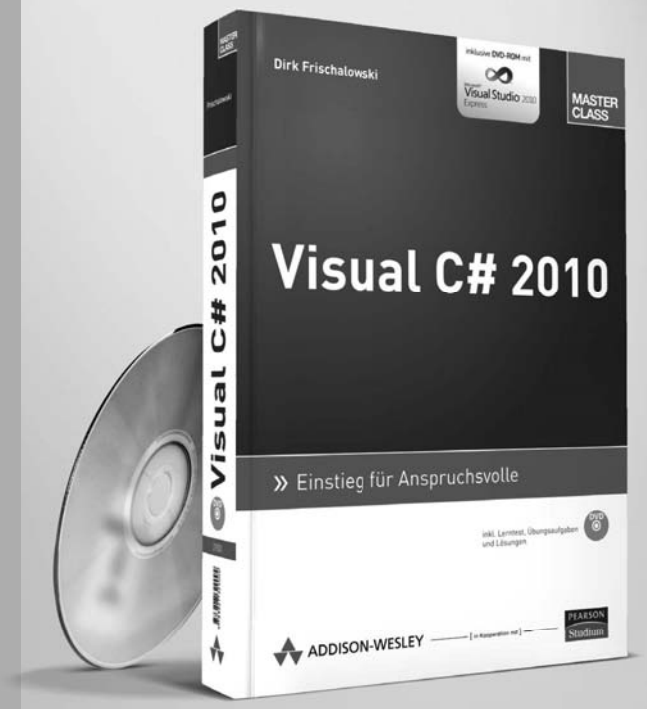
V

W

X

Z

MASTER CLASS



VISUAL C# 2010

Dirk Frischalowski
ISBN 978-3-8273-2900-4
29.80 EUR [D], 30.60 EUR [A], 47.50 sFr*
576 Seiten
http://www.awl.de/2900

Das vorliegende Buch vermittelt Ihnen zunächst das Basiswissen, um die Programmiersprache C# optimal zu nutzen. Dazu gehört eine fundierte Einführung in die Sprachgrundlagen von C#. Danach werden verschiedene Einsatzgebiete des .NET Frameworks behandelt, z.B. die Arbeit mit Dateien, dem Netzwerk oder die Erstellung grafischer Anwendungen (mit Windows Forms). Durch die Übungen an jedem Kapitelende und die Lösungen auf der beiliegenden DVD können Sie das Buch hervorragend zur Überprüfung des Lernfortschritts und zum Einsatz in Schulungen nutzen.

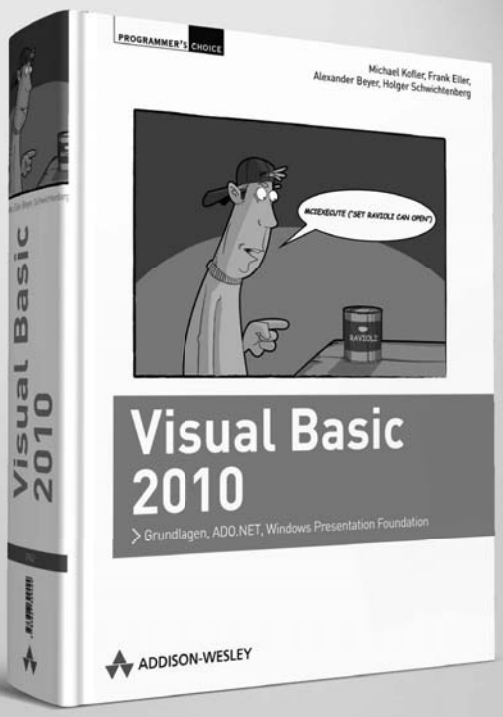
*unverbindliche Preisempfehlung

Mehr Informationen zu Büchern & Video-Trainings auf **www.addison-wesley.de**

TIPP

[The Sign of Excellence] ADDISON-WESLEY

PROGRAMMER'S CHOICE



VISUAL BASIC 2010

Michael Kofler; Frank Eller; Alexander Beyer; Holger Schwichtenberg

ISBN 978-3-8273-2942-4

59.80 EUR [D], 61.50 EUR [A], 93.90 sFr*

1280 Seiten

http://www.awl.de/2942

Das Standardwerk zur Visual-Basic-Programmierung wurde komplett überarbeitet und an die Möglichkeiten von Visual Basic 2010, .NET Framework 4.0 und Visual Studio 2010 angepasst. Mit dem Buch erhalten Sie eine umfassende Einführung in die Syntax von Visual Basic und in die Anwendung wichtiger .NET Bibliotheken einschließlich WPF, LINQ und ADO.NET inklusive Entity Framework. Kompakte Syntaxzusammenfassungen geben eine rasche Referenz wichtiger Schlüsselwörter. Alle Beispielprogramme und das eBook zu "Windows-Forms-Programmierung" (ca. 500 Seiten) finden Sie auf der Website zum Buch.

*unverbindliche Preisempfehlung

Mehr Informationen zu Büchern & Video-Trainings auf **www.addison-wesley.de**

TIPP

[The Sign of Excellence] ADDISON-WESLEY