Mit Visual Basic 2005 Express auf CD

Visual Basic 2005

Windows-Programmierung mit Visual Basic 2005, Visual Studio 2005 und dem .NET Framework 2.0

PETER MONADJEMI

Markt+Technik

KOMPENDIUM

Einführung Arbeitsbuch Nachschlagewerk

PEARSON Education

Markt+Technik

Visual Basic 2005

Windows-Programmierung mit Visual Basic 2005, Visual Studio 2005 und dem .NET-Framework 2.0

PETER MONADJEMI

KOMPENDIUM

Einführung | Arbeitsbuch | Nachschlagewerk

9 Windows Forms für etwas Fortgeschrittene

Es gibt den Maler, der aus der Sonne einen gelben Fleck macht, aber es gibt auch den, der mit Überlegung und Geschick aus einem gelben Fleck eine Sonne macht.

Pablo Picasso (Maler, Grafiker, Bildhauer und visionärer Gestalter von Oberflächen)

Dieses Kapitel richtet sich an jene Leser, die mit dem Prinzip, nach dem ein Button auf einem Formular angeordnet und die Eingabe einer TextBox ausgewertet wird, hinlänglich vertraut sind und nun etwas mehr machen und die restlichen Steuerelemente der Toolbox kennen lernen möchten. Kein Problem, das Windows Forms-Modell hat noch einiges zu bieten. In diesem Kapitel geht es um die etwas fortgeschritteneren Themen, wobei »fortgeschritten« keinesfalls kompliziert bedeutet.

Einer der großen Vorteile des Windows Forms-Modells gegenüber dem alten Modell von VB6 ist nicht, dass dieses einfach nur mehr Features bietet, sondern dass die einzelnen Bestandteile sehr fein aufeinander abgestimmt sind. Da zum Beispiel eine zentrale *Control*-Klasse existiert, von der sich alle Steuerelemente ableiten, verfügen alle Steuerelemente über ein einheitliches Grundgerüst an Eigenschaften und Methoden. Aufgrund der Tatsache, dass eine solche Basisklasse vorhanden ist, können Entwickler ihre eigenen Steuerelemente von dieser Klasse ableiten und diese dadurch mit einem umfangreichen Satz an Grundfunktionalitäten ausstatten. Dieses Prinzip einer modernen Arbeitsteilung findet man an vielen Stellen des Windows Forms-Modells, etwa bei den »Extendern«, die, nachdem sie in ein Formular eingefügt wurden, alle vorhandenen Steuerelemente mit zusätzlichen Eigenschaften ausstatten. Dies spiegelt sich auch in dem Umstand wider, dass eine Toolbar keine Blackbox mehr wie früher darstellt, sondern mit regulären Controls arbeitet und damit sehr flexibel und vor allem auf vertraute Weise erweiterbar ist. In diesem Kapitel geht es ganz konkret um die etwas fortgeschritteneren Steuerelemente, wie Menüs, Toolbars oder TreeViews, mit denen sich anspruchsvollere Benutzeroberflächen realisieren lassen.

Die Themen für dieses Kapitel:

- Menüs
- Symbolleisten
- Speziellere Steuerelemente
- Ziehen-und-Ablegen-Operationen
- Layout ist mehr als nur das Anordnen von Steuerelementen

- Visual Styles bei Windows XP und das Windows-Anwendungsframework
- Es gibt sie doch – die Steuerelementfelder
- MDI-Fenster
- Ein Ersatz für das fehlende Chart-Control

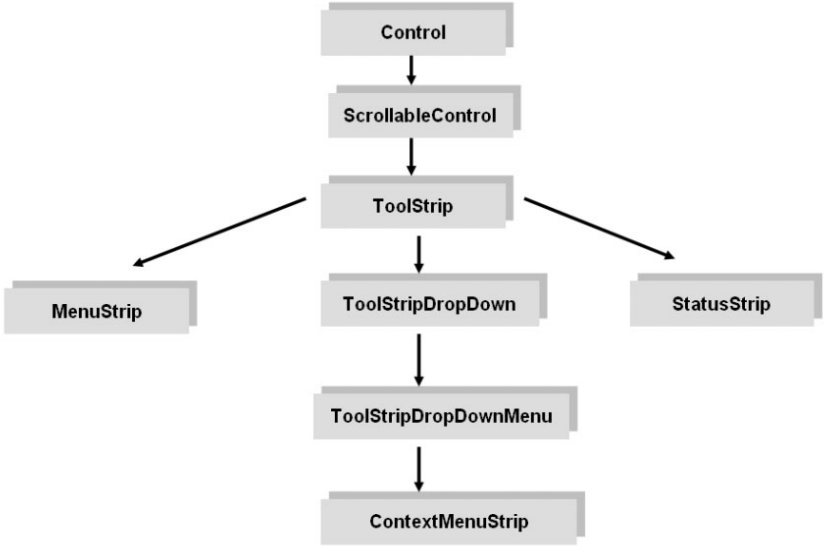
9.1 Menüs

Menüs sind fester Bestandteil praktisch aller modernen Benutzeroberflächen, wenngleich sie oft nur eine Alibifunktion haben, da die wirklich wichtigen Funktionen einfacher über Tastaturshortcuts aufgerufen oder in der Symbolleiste angeboten werden, sodass sich kaum jemand die Mühe macht, dafür ein Menü zu öffnen. Menüs sind bei Visual Basic 2005 schnell zusammengestellt, und sie sehen dabei deutlich attraktiver aus als in der Vergangenheit. Den guten, alten Menüeditor von VB6 gibt es schon lange nicht mehr, Menüs werden direkt im Formular zusammengebaut und in einem komfortablen Elementauflistungseditor nachbearbeitet. Dieser Editor wird z.B. über das Kontextmenü des Controls geöffnet.

Neu bei 2.0:

Mit .NET 2.0 wurden die Steuerelemente MenuStrip *und* ContextMenuStrip *eingeführt, die ein wenig mehr Möglichkeiten und eine etwas modernere Optik bieten. Die folgenden Erklärungen beziehen sich auf diese neuen Controls – die Steuerelemente* MainMenu *und* ContextMenu *aus .NET 1.1 stehen aber natürlich nach wie vor zur Verfügung.*

Abbildung 9.1: Die Hierarchie der neuen Strip-Controls



Die Anatomie eines Menüs

Hinter einer Menüleiste steht ein *MenuStrip*-Control, das Sie wie jedes andere Steuerelement auf das Formular ziehen. Auch wenn sich Menüs komplett per Quellcode aufbauen lassen, ist die Umsetzung im Entwurfsmodus sehr praktisch, da sich die Menüstruktur einfach »herunterschreiben« lässt. Hinter jedem Eintrag der Menüleiste steht ein *ToolStripMenuItem*-Objekt. Ein HIER EINGEBEN deutet an, dass sich ein Eintrag der gewünschten Beschriftung machen lässt. Bezüglich der Benennung haben sich die Entwickler bei Visual Basic 2005 eine nette Kleinigkeit einfallen lassen, indem einem Menüeintrag die gewählte Überschrift vorausgeht. Trägt ein Eintrag beispielsweise die Beschriftung »Datei«, lautet der dazugehörige Name »DateiToolStripMenuItem« usw. Lassen Sie sich nicht durch die verschiedenen HIER EINGEBEN-Einträge irritieren, diese verschwinden wieder, wenn Sie hier nichts eingeben.

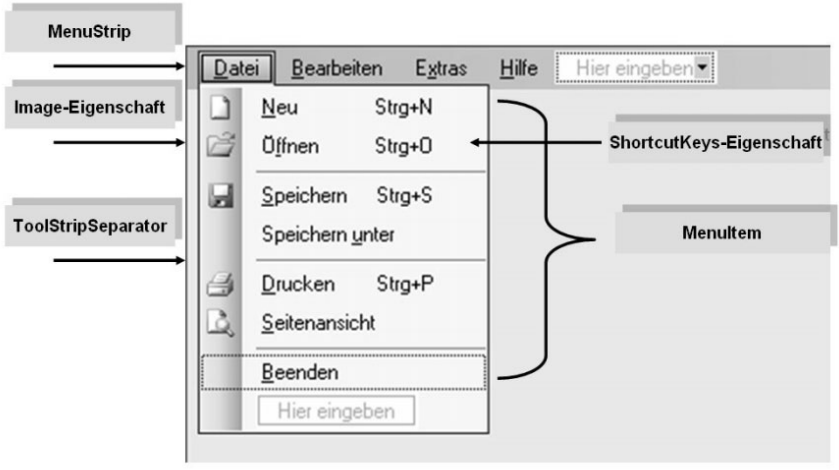


Abbildung 9.2: Die Elemente eines MenuStrip-Menüs

Benötigen Sie ein Menü mit den typischen Standardeinträgen, sollten Sie auf den Eintrag STANDARDELEMENTE EINFÜGEN im Eigenschaftenfenster des MenuStrip-*Controls klicken und sich angenehm überraschen lassen.*

:-) TIPP

Die MenuItem-Klasse

Hinter jedem Menüeintrag steht ein Objekt der *MenuItem*-Klasse, deren wichtigste Mitglieder in Tabelle 9.1 zusammengestellt sind. Ein *MenuItem*-Objekt kann sowohl in ein *MenuStrip*- bzw. *MainMenu*-Control als auch in ein *ContextMenuStrip*- bzw. *ContextMenu*-Control »eingehängt« werden.

Abbildung 9.3: Menüs sind schnell zusammengestellt und wirken optisch auf der Höhe der Zeit.

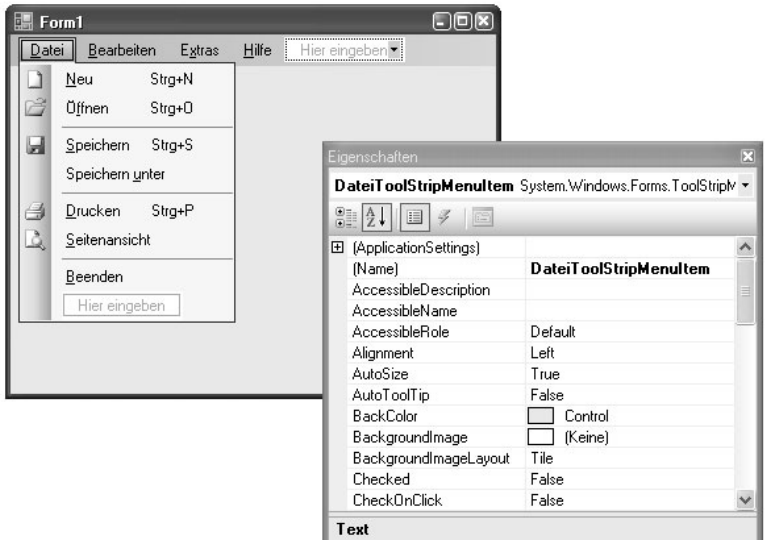


Tabelle 9.1: Interessante Mitglieder der MenuItem-Klasse

Eigenschaft	Bedeutung
Alignment	Legt die Ausrichtung des Eintrags (*links* oder *rechts*) fest.
AutoToolTip	*True*, wenn der Text der *ToolTipText*-Eigenschaft angezeigt werden soll.
Checked	*True*/*False*-Wert, über den sich festlegen lässt, ob der Eintrag mit einem Häkchen versehen wird.
CheckOnClick	*True*, wenn ein mit einem Klick aktiviertes Element diesen Zustand mit einem Häkchen anzeigen soll.
DefaultItem	*True*-/*False*-Wert, der festlegt, ob der Menüeintrag das Standardmenüelement ist, oder die aktuelle Einstellung zurückgibt.
Enabled	*True*-/*False*-Wert, der festlegt, ob der Eintrag ausgewählt werden kann, oder die aktuelle Einstellung zurückgibt.
Index	Gibt die Reihenfolge des Eintrags innerhalb des Menüs zurück oder legt diese fest.
IsParent	*True*-/*False*-Wert, der angibt, ob der Menüeintrag weitere Untereinträge besitzt.
MdiList	*True*-/*False*-Wert, über den sich festlegen lässt, ob dieser Menüeintrag in einem MDI-Hauptfenster die Liste der Kindfenster anzeigt.
MdiListItem	Gibt das *MenuItem*-Objekt zurück, das in einem MDI-Hauptfenster die Liste mit den MDI-Kindfenstern anzeigt.
MenuItems	Steht für die Auflistung der *MenuItem*-Objekte, falls es sich um ein Untermenü handelt.
Mnemonic	Gibt das Zeichen zurück, das auf das &-Zeichen folgt, oder »0«, falls kein Zeichen festgelegt wurde.

Eigenschaft	Bedeutung
OwnerDraw	*True*-/*False*-Wert, über den eingestellt oder abgefragt wird, ob sich der Menüeintrag selbst zeichnet. In diesem Fall wird ein *DrawItem*-Ereignis ausgelöst.
RadioCheck	*True*-/*False*-Wert, über den sich festlegen lässt, ob der Eintrag mit einem Radiobutton anstelle eines Häkchens versehen wird.
Shortcut	Legt die Tastenkombination fest, über die der Menüeintrag aktiviert werden kann, oder gibt die aktuelle Einstellung zurück.
ShowShortcut	*True*-/*False*-Wert, über den sich festlegen fest, ob der Shortcut angezeigt wird oder nicht.
Text	Legt den Text fest, der in dem Eintrag angezeigt wird, oder gibt diesen zurück.

Tabelle 9.1: Interessante Mitglieder der MenuItem-Klasse (Forts.)

Gegenüber der eher schlichten *MenuItem*-Klasse von .NET 1.1 wurde die *ToolStripMenuItem*-Klasse von .NET 2.0 stark erweitert. Die wichtigste Neuerung ist natürlich die *Image*-Eigenschaft, mit der es (endlich) möglich wird, jedem Menüeintrag direkt eine Bitmap zuzuweisen. Auch die *Font*-Eigenschaft ist sehr nett, denn damit kann ein Menü (endlich) die zur Auswahl stehenden Schriftarten so anbieten, dass jeder Eintrag in der entsprechenden Schriftart angezeigt wird[1].

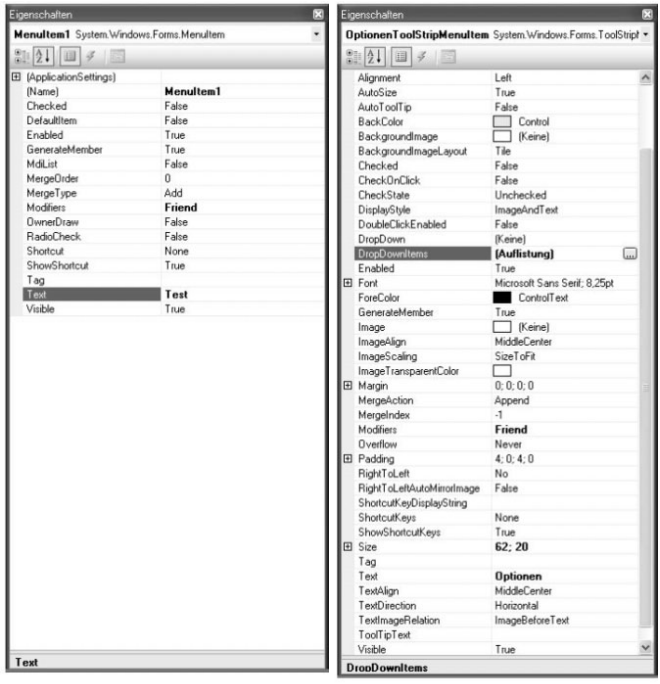


Abbildung 9.4: Die ToolStripMenuItem-Klasse von .NET 2.0 wurde gegenüber der MenuItem-Klasse von .NET 1.1 stark erweitert.

1 Dies ist eine jener Kleinigkeiten, die mir beim Apple Macintosh im Jahre 1985 aufgefallen sind.

Menüereignisse

Die Zeiten sind zum Glück vorbei, in denen ein Menüeintrag lediglich einen harmlosen *Click*-Event lieferte und zum Beispiel das Bewegen des Mauszeigers über einen Eintrag nur per »Subclassing« festgestellt werden konnte. Die *ToolStripMenuItem*-Klasse unterstützt insgesamt 25 Events, wobei neben *Click* und *DoubleClick* die Events *DropDownOpening*, *DropDownOpened* und *MouseHover* erwähnenswert sind. Damit lässt sich praktisch auf jede Aktion im Zusammenhang mit einer Menüauswahl reagieren. In die Ereignisprozedur eines Menüeintrags gelangen Sie wie üblich auf drei verschiedene Arten:

- durch Doppelklick auf den Eintrag
- über die Events im Eigenschaftenfenster
- durch Auswahl des Menüeintrags aus der linken Auswahlliste des Programmcodefensters

Kontextmenüs

Ein Kontextmenü ist ein Menü, das nicht von Anfang an sichtbar ist, sondern in der Regel durch Anklicken eines Steuerelements oder der Innenfläche des Formulars geöffnet wird. Ein Kontextmenü steht in der Toolbox als eigener Eintrag (CONTEXTMENUSTRIP) zur Verfügung. Ziehen Sie es auf ein Formular, erscheint es zwar unterhalb der Titelleiste und lässt sich wie ein reguläres Menü erstellen, es wird aber anschließend wieder ausgeblendet, da es während der Programmausführung zunächst ebenfalls unsichtbar ist. Möchten Sie es im Entwurfsmodus erneut anzeigen, klicken Sie einfach das Symbol im Komponentenfach unterhalb des Formulars an.

Damit ein Kontextmenü sichtbar wird, muss dessen *Show*-Methode aufgerufen werden, der die Position übergeben wird, an der das Menü erscheinen soll. Da ein Kontextmenü im Allgemeinen über einen Klick mit der *rechten* Maustaste geöffnet wird, erfolgt der Aufruf im *MouseDown*-Event eines Formulars oder eines Steuerelements.

Hier ein kleines Beispiel für die Anzeige des Kontextmenüs ContextMenuStrip1 *im* MouseDown-*Event des Formulars:*

```
Private Sub Form1_MouseDown(ByVal sender As Object, ByVal e _
    As System.Windows.Forms.MouseEventArgs) Handles Me.MouseDown
  If e.Button = Windows.Forms.MouseButtons.Right Then
    ContextMenuStrip1.Show(Me.PointToScreen(e.Location), _
      ToolStripDropDownDirection.BelowRight)
  End If
End Sub
```

Achten Sie auf die Umrechnung der Mauskoordinaten über die *PointToScreen*-Methode des Formulars. Da *Show* Bildschirmkoordinaten erwartet, *MouseDown* aber die Koordinaten bezogen auf das Formular liefert, müssen diese entsprechend konvertiert werden. Sehr elegant ist auch die Möglichkeit, über die *Show*-Methode festzulegen, in welche Richtung sich das Menü bezogen auf die Koordinaten öffnen soll. .NET 2.0 überzeugt durch viele dieser netten Kleinigkeiten.

Kontextmenüs direkt zuweisen

Noch ein wenig einfacher geht das Zuordnen eines Kontextmenüs über die Eigenschaften *ContextMenu* bzw. *ContextMenuStrip* eines Formulars oder Steuerelements. In diesem Fall wird das Menü standardmäßig über die rechte Maustaste geöffnet, und Sie müssen den *MouseDown*-Event nicht bemühen:

```
Me.ContextMenuStrip = ContextMenuStrip1
```

Sich selbst zeichnende Menüeinträge

Was bei früheren Visual Basic-Versionen gewisse API-Klimmzüge erforderte und daher in der Regel für Visual Basic-Programmierer nicht infrage kam, lässt sich seit Visual Basic .NET etwas leichter bewerkstelligen. Besitzt die *OwnerDraw*-Eigenschaft eines *MenuItem*-Objekts, das heißt eines einzelnen Menüeintrags, den Wert *True*, wird vor der Anzeige des Eintrags die *DrawItem*-Ereignisprozedur aufgerufen. Dabei besteht die Gelegenheit, im Prinzip beliebige Zeichenoperationen auszuführen. Zwar wird dieser Ereignisprozedur ein *Graphics*-Parameter übergeben, der für die zur Verfügung stehende Zeichenfläche steht, doch ist es leider nicht damit getan, einfach die *DrawIcon*-Methode aufzurufen, um ein Icon zu zeichnen. Der Grund liegt darin, dass die Zeichenfläche auch eine Höhe erhalten muss, die von dem zu zeichnenden Icon und der anzuzeigenden Schriftart abhängt. Da Menüeinträge bei .NET 2.0 nicht nur Bitmaps anzeigen können, sondern sich auch die Schriftarten individuell einstellen lassen, dürfte es für sich selbst zeichnende Menüs nur wenige Notwendigkeiten geben. Ein Beispiel für ein OwnerDraw-Control erhalten Sie in Kapitel 8 am Beispiel einer ListBox, die jeden Eintrag in einer eigenen Schriftfarbe anzeigt.

9.2 Symbolleisten – das neue ToolStrip-Control

Symbolleisten gehören wie die Menüs einfach zu einer modernen Benutzeroberfläche. Leider hatten gerade Visual Basic-Programmierer in den letzten Jahren nicht allzu viel Spaß mit Symbolleisten. Das ist spätestens mit Visual Basic 2005 vorbei, denn das neue *ToolStrip*-Control präsentiert sich vielseitig und vor allem leicht bedienbar. Ordnen Sie es auf einem Formular an, erhalten Sie davon einen ersten Vorgeschmack, denn es zeigt sich nicht nur in einer hübschen und modernen Optik, sondern bietet eine Auswahlliste, über die Sie Elemente zur Symbolleiste hinzufügen. Das Repertoire ist wirklich beeindruckend, denn neben einem Button stehen als weitere Elemente zur Verfügung: Label, SplitButton, DropDownButton, ComboBox, TextBox und der obligatorische Separator. Damit lassen sich wirklich funktionale Symbolleisten erstellen. Fällt Ihnen der kleine Pfeil in der rechten oberen Ecke auf? Nach einem Klick darauf erscheint das übliche Auswahlmenü, das besonders wichtige Aktionen zusammenfasst. Rufen Sie den Eintrag STANDARDELEMENTE EINFÜGEN auf, und lassen Sie sich wieder einmal angenehm überraschen – fertig ist die Symbolleiste mit allen wichtigen Elementen, die ein Anwendungsfenster heutzutage braucht. Die Symbolleiste können Sie selbstverständlich beliebig editieren.

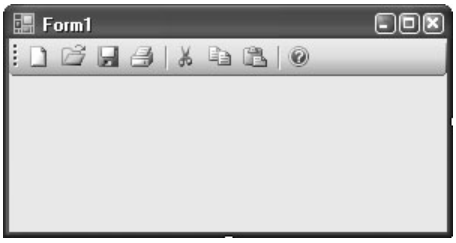


Abbildung 9.5: Wird auf Knopfdruck angelegt – eine Symbolleiste mit Standardelementen.

Weder das ToolStrip- *noch das* MenuStrip-*Control muss am oberen Fensterrand andocken, es stehen verschiedene Dockpositionen zur Auswahl.*

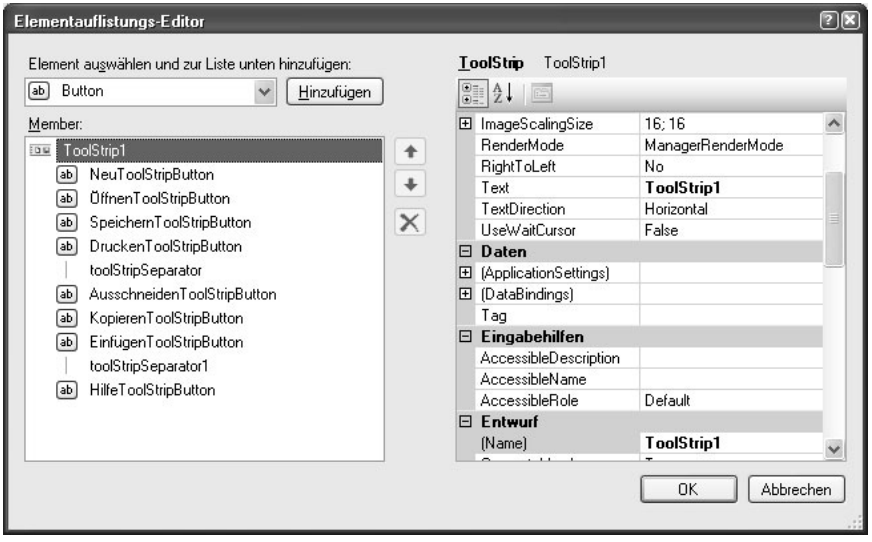


Abbildung 9.6: Im Auflistungs-Editor kann jedes Element der Toolleiste nachträglich bearbeitet werden.

Beispielprojekt Aktiendepotverwaltung

Wer vom (praktisch permanent stattfindenden) Börsenboom profitieren möchte, benötigt als Erstes eine Depotverwaltung (und als Nächstes einen guten Steuerberater). Auch wenn es das natürlich alles online gibt, Selbermachen bereitet zum einen mehr Spaß und bietet zum anderen die Gelegenheit, eine maßgeschneiderte Lösung zu erhalten. Die Aktiendepotverwaltung auf der Buch-CD soll weniger Ihr Vermögen mehren (und vor allem niemanden zu riskanten Spekulationen verleiten), sondern in erster Linie (eigentlich sogar ausschließlich) ein nettes Beispiel für die verschiedenen Windows Forms-Steuerelemente bieten.

Sie finden das Beispiel auf der Buch-CD in der Projektdatei 09_Aktiendepot.sln.

Abbildung 9.7: Eine kleine Depotverwaltung (alle Angaben natürlich ohne Gewähr)

Die Rolle des ToolStrip-Containers

Anwender einer modernen Office-Anwendung sind es gewohnt, dass sie die Befehlsleisten beliebig im Anwendungsfenster verschieben können. Damit dies auch unter einer Windows Forms-Anwendung realisiert werden kann, wurde mit .NET 2.0 das ToolStripContainer-Control eingeführt. Es stellt eine während der Programmausführung unsichtbare Fläche dar, innerhalb derer die verschiedenen Strip-Controls an den vier Rändern des Containers angedockt werden können. Neben der zentralen Fläche in der Mitte, repräsentiert durch ein *ContentPanel*, wird jeder Rand durch ein eigenes Panel-Control (etwa *TopToolStripPanel*) repräsentiert. Um ein ToolStrip-Control in einen Container einzubetten, ist es am einfachsten, aus dessen Aufgabenmenü den Eintrag IN TOOLSTRIPCONTAINER EINBETTEN aufzurufen, woraufhin dann der Container nachträglich hinzugefügt wird.

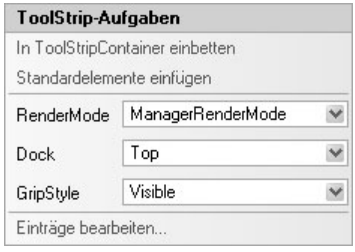


Abbildung 9.8: Ein ToolStrip-Control wird nachträglich in einen ToolStripContainer eingebettet.

Der Umgang mit dem ToolStrip-Container ist am Anfang ein wenig gewöhnungsbedürftig, da sich das Control gegen das Selektieren mit der Maus ein wenig sträubt und es nicht ganz einfach ist, ein hinzugefügtes *ToolStripContainer*-Control wieder zu entfernen (es muss dazu nicht nur leer sein, es darf sich auch kein anderes *ToolStripContainer*-Control mit einem Inhalt auf dem Formular befinden). Auch das Herausnehmen eines Controls aus dem »Einflussbereich« eines Containers scheint nicht sehr gut zu funktionieren. Dazu ein Tipp, der schnell übersehen wird: Klicken Sie die Innenfläche eines Steuerelements mit der rechten Maustaste an, und wählen Sie zunächst AUSWÄHLEN und anschließend das gewünschte Control. Dadurch lässt sich ein Control in der Regel am einfachsten selektieren.

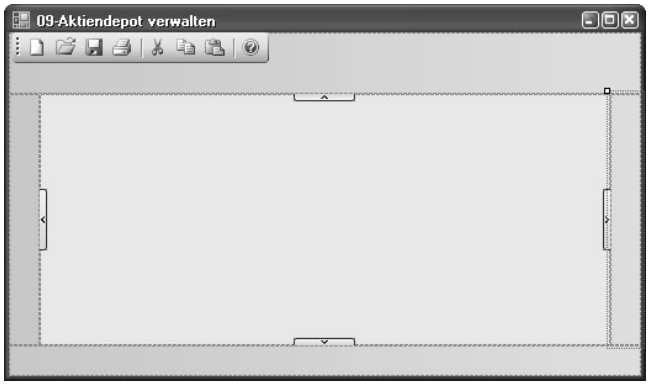


Abbildung 9.9: Die Symbolleiste kann dank des ToolStripContainers flexibel angedockt werden.

Unter Umständen ist es einfacher, den Designer zu ignorieren und eine Toolleiste inklusive des umgebenden Containers im Code anzuordnen:

```
Dim toolStripContainer1 = New ToolStripContainer()
Dim toolStrip1 = New System.Windows.Forms.ToolStrip()
toolStrip1.Items.Add("1")
toolStrip1.Items.Add("2")
toolStrip1.Items.Add("3")
toolStripContainer1.TopToolStripPanel.Controls.Add(toolStrip1)
Controls.Add(toolStripContainer1)
```

Diese Befehle legen eine Toolleiste mit drei Einträgen an, die in einen Container eingebettet wird.

9.3 Speziellere Steuerelemente

In diesem Abschnitt werden die etwas spezielleren Steuerelemente vorgestellt, die Sie am Anfang seltener benötigen werden, die aber ebenfalls zu den Stammsteuerelementen gehören.

Die StatusStrip-Klasse

Das ebenfalls neu mit .NET 2.0 eingeführte StatusStrip-Control ist für die Statusleiste zuständig, die jedem Formular zugeordnet werden kann. Anders als in der Vergangenheit, sprich in der »VB6-Ära«, bietet die Statusleiste einiges an Möglichkeiten:

- Neben einer simplen TextBox stehen auch ein Fortschrittsbalken und eine DropDown-Liste zur Auswahl.
- Es können auch Hyperlinks angeordnet werden.
- Es kann dem Anwender erlaubt werden, die Anordnung der Elemente während der Programmausführung bei gedrückter Alt-Taste zu ändern. Dazu ist die *AllowItemReorder*-Eigenschaft auf *True* zu setzen.

Insbesondere die Möglichkeit, Auswahllisten, Fortschrittsbalken und Links anzuzeigen, macht die neue Statusbar sehr attraktiv.

Das Bearbeiten des StatusStrip-Controls und der dazugehörigen Einträge geschieht am besten über den Elementauflistungs-Editor.

Tabelle 9.2: Ein paar interessante Eigenschaften der StatusStrip-Klasse

Eigenschaft	Bedeutung
AllowItemReorder	*True*, wenn die Elemente vom Anwender in der Reihenfolge vertauscht werden können.
LayoutStyle	Ermöglicht es, die Elemente nicht nur nebeneinander, sondern zum Beispiel auch überlappend darzustellen. Die beste Einstellung ist *Table*, hier besitzt jedes Element seinen festen Bereich. Bei *Flow* folgen die Elemente, je nach aktueller Breite, unmittelbar aufeinander.
TextDirection	Legt die Ausrichtung des Textes fest, wobei der Text auch um 90 oder 270 Grad gedreht, also vertikal angezeigt werden kann.

Die optimale Breite der einzelnen Elemente festzulegen ist nicht ganz einfach. Durch Setzen der Spring*-Eigenschaft der* ToolStripStatusLabel*-Klasse auf* True *kann erreicht werden, dass ein Element den noch verbleibenden Platz komplett ausfüllt.*

Anordnen eines Fortschrittsbalken in der Statusleiste

Im Folgenden soll an einem kleinen Beispiel nicht nur gezeigt werden, wie sich (größere) Dateien asynchron laden lassen, sondern auch, wie der aktuelle Stand beim Laden per Fortschrittsbalken in der Statusleiste visualisiert werden kann.

Sie finden das Beispiel auf der Buch-CD in der Projektdatei 09_Progressbar.sln.

Beispiel

*Das folgende Beispiel durchsucht ein Verzeichnis mit aus Outlook (oder einem anderen E-Mail-Programm) heraus gespeicherten E-Mail-Nachrichten Zeile für Zeile und zeigt sowohl den Namen der gerade durchsuchten Datei, die Zeile sowie den Fortschritt mithilfe zweier Labels (*tslbDateiname *und* tslbDateizeile*) und eines Fortschrittsbalkens (*tspbDateien*) in der Statusbar an. Damit bei wenigen Dateien überhaupt etwas zu sehen ist, wird nach jeder Zuweisung die* Refresh*-Methode aufgerufen und die Programmausführung durch die* Threading.Thread.Sleep*-Methode ein wenig »abgebremst«.*

```
With tspbDateien
  .Minimum = 0
  .Maximum = New DirectoryInfo(MsgPfad).GetFiles("*.txt").Length
End With
```

```
For Each Fi As FileInfo In New DirectoryInfo(MsgPfad).GetFiles("*.txt")
  AnzahlDateien += 1
  AnzahlZeilen = 0
  tspbDateien.Value = AnzahlDateien
  tslbDateiname.Text = Fi.Name.Substring(0, Math.Min(Fi.Name.Length, _
    40))
  StatusStrip1.Refresh()
  Threading.Thread.Sleep(1000)
  Dim Sr As New StreamReader(Fi.FullName)
  Do While Sr.Peek <> -1
    Dim Zeile As String = Sr.ReadLine
    ' Hier müsste die Zeile untersucht werden
    AnzahlZeilen += 1
    tslbDateizeile.Text = AnzahlZeilen
    StatusStrip1.Refresh()
  Loop
  Sr.Close()
  Threading.Thread.Sleep(1000)
Next
Threading.Thread.Sleep(1000)
tspbDateien.Value = 0
```

Abbildung 9.10: Die Statusleiste zeigt zwei Labels und einen Fortschrittsbalken an.

Daten in einem Baum anordnen – das TreeView-Steuerelement

Das TreeView-Control zeigt einen aus Knoten bestehenden Baum von Textelementen an, wobei jedem Knoten eine kleine Bitmap zugeordnet werden kann. Jeder Knoten wird durch ein *TreeNode*-Objekt repräsentiert. Es besitzt selbst eine *Nodes*-Auflistung, an die weitere *TreeNode*-Objekte gehängt werden können, sodass sich eine theoretisch beliebig tief verschachtelte Baumstruktur realisieren lässt. Ein *TreeNode*-Objekt besteht aus einem Text und einem Icon. Letzteres kann leider nicht direkt zugeordnet werden, sondern muss aus einer *ImageList* stammen. Die Zuordnung geschieht über einen Index oder ein Schlüsselwort. Über verschiedene Eigenschaften der *TreeNode*-Klasse kommt man schnell an die Nachbarknoten heran. Der Umgang mit dem TreeView-Steuerelement ist relativ einfach, wie das folgende Beispiel zeigt.

Ein neuer Knoten wird wie folgt angelegt:

```
Dim Tn As TreeNode = TreeView1.Nodes.Add("Neuer Eintrag")
```

Die Variable *Tn* steht für den neuen Knoten, sodass der folgende Befehl einen Unterknoten anhängt:

```
Dim SubTn As TreeNode = Tn.Nodes.Add("Ein Untereintrag")
```

Soll der Unterknoten nicht weiter angesprochen werden, muss der Rückgabewert von *Add* natürlich keiner Variablen zugewiesen werden.

Sie finden das Beispiel auf der Buch-CD in der Projektdatei 09_TreeViewBsp.sln.

Das Beispiel listet alle Dateien mit der Erweiterung .doc *im Verzeichnis* Eigene Dateien *als Knoten auf und fügt verschiedene Dokumenteigenschaften als Unterknoten hinzu. Jeder Dokumentknoten wird durch ein kleines Icon dargestellt, das aus einem ImageList-Control stammt. Anders als die mit .NET 2.0 eingeführten Strip-Controls ist dies die einzige Möglichkeit, um einem TreeView- oder einem ListView-Control Icons zuzuordnen.*


```
Dim DocPfad As String = Environment.GetFolderPath(Environment. _
  SpecialFolder.MyDocuments)
Dim Dso As Object = CreateObject("DSOleFile.PropertyReader")
Dim DsoProps As Object
For Each Fi As FileInfo In New DirectoryInfo(DocPfad).GetFiles("*.doc")
  DsoProps = Dso.GetDocumentProperties(Fi.FullName)
  Dim Tn As TreeNode = TreeView1.Nodes.Add(Fi.Name, Fi.Name, 0)
  Tn.Nodes.Add("Autor", "Autor: " & DsoProps.Author, 1)
  Tn.Nodes.Add("AnzahlWorte", "Anzahl Worte: " & DsoProps.WordCount, 1)
  Tn.Nodes.Add("Dokumentversion", "Dokumentversion: " & _
    DsoProps.RevisionNumber, 1)
Next
```

Abbildung 9.11: Das TreeView-Control zeigt ein Baumstruktur an.

Kurzer Exkurs: Eigenschaften in Office-Dokumenten lesen

Wie lassen sich die zahlreichen Eigenschaften eines Microsoft Office-Dokuments wie der Name des Autors, die Anzahl der Seiten, das Datum der letzten Änderung usw. lesen? Entweder indem ein Programm per COM-Interop auf die jeweilige

Office-Anwendung zugreift, die das Dokument lädt und die einzelnen Eigenschaften über die *BuiltInDocumentProperties*-Eigenschaft ausliest. Oder über eine kleine COM-Dll mit dem Namen *DsoFile.dll*, die von der Microsoft-Website heruntergeladen und registriert werden muss, sodass sie jeder Anwendung zur Verfügung steht.

Das folgende Beispiel liest die Author*-Eigenschaft eines Word-Dokuments, das über das* FileInfo*-Objekt* Fi *angesprochen wird.*

```
Dim Dso As Object = CreateObject("DSOleFile.PropertyReader")
Dim DsoProps As Object
DsoProps = Dso.GetDocumentProperties(Fi.FullName)
Autor = DsoProps.Author
```

Anstelle von CreateObject *kann auch über* PROJEKT/VERWEIS HINZUFÜGEN *ein Verweis auf die Datei* Dsofile.dll *eingebunden und die Variable wie folgt instanziiert werden :*

```
Dim Dso As New DSOleFile.PropertyReader
```

Mehr zu dem Thema finden Sie in dem Knowledgebase-Artikel http://www.microsoft.com/germany/technet/datenbank/articles/600696.mspx.

Wie erhält ein TreeView-Knoten sein Icon?

Voraussetzung dafür, dass einem TreeView-Knoten ein Icon zugeordnet werden kann, ist eine ImageList-Komponente, die ebenfalls über die Toolbox angeboten wird. Die einzige Aufgabe einer ImageList-Komponente ist es, anderen Steuerelementen (im Wesentlichen nur ListView und TreeView, weil das Toolbar-Control von .NET 1.1 in .NET 2.0 durch das ToolStrip-Control abgelöst wurde) Bitmaps zur Verfügung zu stellen. Da die ImageList zur Laufzeit unsichtbar ist, ist sie als Komponente und nicht als Control implementiert und wird nach dem Anordnen auf dem Formular im »Komponentenfach« angezeigt. Der Umgang mit der ImageList besteht aus folgenden Schritten:

1. Hinzufügen von Bitmaps über die *Images*-Eigenschaft. Dabei kommt es vor allem auf die richtige Größe an. Auch wenn die ImageList in diesem Punkt nicht wählerisch ist, eine große Bitmap mit einer Auflösung von zum Beispiel 640 × 480 Pixel oder höher sieht in einem TreeView im Allgemeinen nicht besonders vorteilhaft aus, wenn sie auf 32 x 32 oder 16 x 16 Pixel heruntergeskaliert wurde. Sehr praktisch ist, dass die Größe der Bitmap, die einem angeschlossenen Steuerelement zur Verfügung gestellt wird, in der Aufgabenliste eingestellt werden kann.
2. Dieser Schritt ist optional: das Festlegen einer transparenten Farbe. Viele Icons, insbesondere jene, die Teil von Visual Studio 2005 sind, enthalten eine Hintergrundfarbe (zum Beispiel Lila, genauer gesagt Fuchsia mit dem RGB-Code 255, 0, 255), die über die Eigenschaft *TransparentColor* als transparente Farbe ausgewählt werden muss, damit der Hintergrund entsprechend durchsichtig wird. Wenn die Hintergrundfarbe nicht bekannt ist, bietet es sich an, diese mit einem Bildbearbeitungsprogramm zu ermitteln, da ein Erraten des richtigen Farbcodes nur schwerlich möglich ist.

3. Die ImageList wird dem ListView- oder TreeView-Steuerelement über deren Eigenschaften *LargeImageList* und *SmallImageList* bzw. *ImageList* zugeordnet.
4. Das Icon wird über seine Indexnummer angesprochen:

```
Dim Tn As TreeNode = TreeView1.Nodes.Add(Fi.Name, Fi.Name, 0)
```

Es ist kein Problem, Bitmaps nachträglich zu ändern, wenn die ImageList bereits einem Control zugeordnet wurde.

Auf Ereignisse des TreeView reagieren

Ein NodeSelect-Ereignis gibt es beim TreeView-Control nicht. Das Ereignis, das diesem am nächsten kommt, ist der *AfterSelect*-Event, der ausgelöst wird, nachdem ein Knoten selektiert wurde. Über den *TreeViewEventArgs*-Parameter wird der selektierte Knoten übergeben.

Oft soll eine Aktion nur für Knoten ausgeführt werden, die sich eine Ebene höher in der Knotenhierarchie befinden. In diesem Fall lässt sich über die Parent*-Eigenschaft feststellen, ob der Knoten einen »Elternknoten« besitzt oder, sollte das nicht der Fall sein, ein oberster Knoten ist. Sehr praktisch ist auch die universelle* Tag*-Eigenschaft, der beliebige Werte zugewiesen werden können, unter anderem eine Information, die etwas über die Position des Knotens aussagt. Eine* Level*-Eigenschaft gibt es bei der* TreeNode*-Klasse auch.*

Tabelle 9.3: Interessante Mitglieder der TreeView-Klasse

Mitglied	Bedeutung
CheckBoxes-Eigenschaft	Legt fest, ob jeder Knoten durch eine CheckBox dargestellt wird. Sehr interessante Eigenschaft.
CollapseAll-Methode	Klappt alle Knoten auf einmal zu.
ExpandAll-Methode	Klappt alle Knoten auf einmal auf.
GetNodeCount-Eigenschaft	Gibt die Anzahl der Knoten des aktuellen Knotens zurück.
HideSelection-Eigenschaft	Legt fest, ob die aktuelle Auswahl erhalten bleibt, wenn das Steuerelement den Fokus verliert.
HotTracking-Eigenschaft	Legt fest, ob ein Knotenname als Hyperlink dargestellt wird, wenn der Mauszeiger darauf positioniert wird. Sehr interessante Eigenschaft.
ImageIndex-Eigenschaft	Steht für den Index des mit dem Knoten angezeigten Icons und bezieht sich auf ein ImageList-Steuerelement.
ImageKey-Eigenschaft	Erlaubt das einem Knoten zugeordnete Icon über einen Schlüsselnamen auszuwählen.
ImageList-Eigenschaft	Legt das ImageView-Steuerelement fest, das die Icons zur Verfügung stellt.

Tabelle 9.3: Interessante Mitglieder der TreeView-Klasse (Forts.)

Mitglied	Bedeutung
LabelEdit-Eigenschaft	*True*, wenn der Knotenname bearbeitet werden kann.
Nodes-Eigenschaft	Steht für die Unterknoten des Knotens als *Nodes*-Auflistung.
SelectedNode-Eigenschaft	Steht für den aktuell selektierten Knoten.
Sort-Methode	Sortiert die Knotennamen alphabetisch.
TopNode-Eigenschaft	Steht für den ersten sichtbaren Knoten im TreeView.
TreeViewNodeSorter-Eigenschaft	Steht für einen Vergleicher, der *IComparer* implementiert, um die Knotenelemente sortieren zu können.

Tabelle 9.4: Interessante Mitglieder der TreeNode-Klasse

Mitglied	Bedeutung
Checked-Eigenschaft	*True*, wenn das Element selektiert ist (wenn die *CheckBoxes*-Eigenschaft des TreeView entsprechend gesetzt ist).
Collapse-Methode	Klappt alle Unterknoten des Knotens zu. Per *True*-Parameter behalten die Kindknoten ihren aktuellen Zustand.
EnsureVisible-Methode	Zeigt den Baum so an, dass dieser Knoten sichtbar ist.
ExpandAll-Methode	Erweitert den Knoten, sodass alle Kindknoten sichtbar werden.
FirstNode-Eigenschaft	Steht für den ersten Unterknoten.
FullPath-Eigenschaft	Kompletter Pfad des Knotens.
GetNodeCount-Methode	Gibt die Anzahl der Unterknoten zurück.
IsSelected-Eigenschaft	*True*, wenn der Knoten aktuell selektiert ist.
LastNode-Eigenschaft	Steht für den letzten Unterknoten.
Level-Eigenschaft	Steht für die Ebene des Knotens innerhalb der Knotenhierarchie.
Nodes-Eigenschaft	Steht für die Unterknoten als *Nodes*-Auflistung.
Parent-Eigenschaft	Steht für den übergeordneten Knoten.
Remove-Methode	Entfernt den Knoten aus der *Nodes*-Auflistung.
Tag-Eigenschaft	Allgemeine Eigenschaft vom Typ *Object*, durch die jedem Knoten etwas Beliebiges, etwa eine ID, zugeordnet werden kann.
Text-Eigenschaft	Der angezeigte Text des Knotens.
Toggle-Methode	Schaltet den Knoten zwischen erweitert und reduziert um.
TreeView-Eigenschaft	Steht für das TreeView-Control, zu dem der Knoten gehört.

Ein Formular beliebig unterteilen – das TabControl-Steuerelement

Ein TabControl-Steuerelement ist eine sehr praktische Angelegenheit, denn es stellt eine beliebige Anzahl von Unterteilungen, auch Registerkarten oder Tabs genannt, zur Verfügung, auf denen Steuerelemente angeordnet werden können. Jede Registerkartenseite wird durch ein *TabPage*-Objekt repräsentiert. Benötigt man eine typische Registerkarte, um auf konstantem Raum prinzipiell beliebig viele Eingabeelemente – unterteilt nach Kategorien – unterzubringen, ist das TabControl dafür bestens geeignet. Auch wenn sich das TabControl, wie jedes Steuerelement, komplett programmgesteuert einrichten lässt, ist es am einfachsten, das Anlegen der Register im Eigenschaftenfenster vorzunehmen.

Abbildung 9.12: Ein TabControl erlaubt das Unterteilen eines Formulars in Register (Tabs).

Mitglied	Bedeutung
DisplayRectangle-Eigenschaft	Steht für die Innenfläche der stets gleich großen Registerseite als *Rectangle*-Struktur.
Multiline-Eigenschaft	*True*, wenn sich die Reiter auf mehrere Zeilen verteilen können – ansonsten wird ein Pfeil zum Scrollen angezeigt.
SelectedTab-Eigenschaft	Repräsentiert die momentan ausgewählte Registerkarte als *TabPage*-Objekt.
TabCount-Eigenschaft	Anzahl der Registerkarten.
TabPages-Eigenschaft	Steht für die Auflistung der Registerseiten in Gestalt einer *TabPageCollection*, die entsprechende Methoden wie *Add*, *Insert* und *Remove* anbietet.

Tabelle 9.5: Interessante Mitglieder der TabControl-Klasse

Das TabControl*-Steuerelement besitzt leider auch in der aktuellen Version 2.0 einen lästigen Bug, der darin besteht, dass manchmal die Register ihre Anordnung ändern. Sie müssen dann in der Designer-Datei des Formulars die Stelle lokalisieren, an der die Seiten über die* Add*-Methode der* Controls*-Auflistung zum TabControl hinzugefügt werden, und die Reihenfolge an dieser Stelle wieder korrigieren. Dadurch wird das Problem aber nur vorübergehend behoben und nicht ganz abgestellt.*

Steuerelemente mit der Find-Methode der Controls-Auflistung finden

Je umfangreicher ein Formular wird, desto nützlicher kann es sein, einzelne Steuerelemente über den Namen lokalisieren zu können. Stellen Sie sich ein TabControl mit zwölf Registern vor, auf dem jedes Register eine ComboBox enthält, die Monate anzeigt. Theoretisch muss die Liste zwölf Mal gefüllt werden. Statt zwölf Aufrufen wäre genau ein Aufruf sinnvoller, dem einfach das zu füllende Control übergeben wird. Soll das Füllen erst dann geschehen, wenn das Register aktiviert wird, muss der Füllroutine jene ComboBox übergeben werden, die sich auf dem derzeit selektierten Register befindet. Da dieses Register unter Umständen mehrere Steuerelemente und auch mehrere ComboBoxen enthält, wäre eine Suchfunktion sehr praktisch. Diese existiert in Gestalt der *Find*-Methode der *Controls*-Auflistung. Diese Methode lokalisiert ein Control anhand des dazugehörigen Namens und gibt das Control zurück.

Sie finden das Beispiel auf der Buch-CD in der Projektdatei 09_TabFindControl.sln.

Das folgende Beispiel gibt die (einzige) ComboBox zurück, die sich auf der momentan selektierten Registerkarte im TabControl tbMaerkte *befindet, wobei lediglich der Name der ComboBox bekannt ist. Wurde diese gefunden, wird sie einer Prozedur als Parameter übergeben:*

```
Try
  Dim cbTemp As Control = tbMaerkte.SelectedTab.Controls.Find _
    ("cbMonateRH" & CStr(tbMaerkte.SelectedTab.Tag), False)(0)
  MonatsListeFuellen(CType(CbTemp, ComboBox))
Catch ex As SystemException
  ' Hier passiert nichts
End Try
```

Ist der Name nicht bekannt, kann die *Controls*-Auflistung auch per *For Each* durchsucht werden – und sollten Controls vorhanden sein, die wiederum andere Controls als Container enthalten können, muss die Suche rekursiv erfolgen, was aber nicht weiter kompliziert ist, da dazu bei *Find* lediglich ein *True*-Flag übergeben werden muss.

Die Explorer-Ansicht für beliebige Gegenstände – das ListView-Steuerelement

Das ListView-Steuerelement ist ein alter Bekannter, da es zu den »Windows Common Controls« gehört, die (seit Windows 95) vom Betriebssystem zur Verfügung gestellt werden. Es stellt die typische Innenansicht des Windows-Explorers zur Verfügung, bei der die Elemente wahlweise als große Symbole, als kleine Symbole, in Form einer Listenansicht oder als Detailansicht angezeigt werden können. Eine ListView ist immer dann sehr praktisch, wenn eine mehrspaltige Liste benötigt wird. Da das ListView-Control von .NET 2.0 lediglich das Standardsteuerelement

der Windows-Library kapselt, unterscheidet sich die Programmierung unter Visual Basic 2005 nur geringfügig von der unter VB6. Auch wenn eine ListView insgesamt fünf Ansichten bietet (Details, große Symbole, kleine Symbole, Liste und geschachtelt), spielt für die Praxis meistens nur die Detailansicht eine Rolle, die durch Setzen der *View*-Eigenschaft auf den Wert *Details* aktiviert wird.

Eine ListView erhält ihren Inhalt in Gestalt von Texten, die wie bei einer ListBox über die *Add*-Methode der *Items*-Auflistung hinzugefügt werden. Anders als eine ListBox kann jeder Eintrag Untereinträge besitzen, die über die *Add*-Methode der *SubItems*-Eigenschaft eines *ListViewItem*-Objekts hinzugefügt werden, das von der *Add*-Methode der *Items*-Auflistung zurückgegeben wird. Voraussetzung ist allerdings, dass zuvor eine Spalte zur *Columns*-Auflistung der ListView hinzugefügt wurde.

Sie finden das Beispiel auf der Buch-CD in der Projektdatei 09_ListViewBsp.sln.

Das folgende Beispiel listet MP3-Musikdateien im Verzeichnis Eigene Musik *in einem ListView-Control auf und gibt zusätzlich zur Größe auch die Bitrate und die Länge der Datei an. Da dies allerdings mit den Mitteln der .NET-Klassenbibliothek alleine nicht möglich ist, greift das Programm auf eine Komponente zurück, welche die so genannten ID3-Tags einer MP3-Datei ausliest. Eine solche Komponente ließe sich mit relativ wenig Aufwand selbst programmieren (mehr dazu in Kapitel 11, wenn es um eigene Komponenten geht), Sie finden aber auch viele fertige Lösungen im Internet – das folgende Beispiel bedient sich eines Projektes von* Thommy Mewes*, das Sie unter* http://www.codeproject.com/vb/net/vbmp3header.asp *finden.*


```
Dim MusikPfad As String = Environment.GetFolderPath _
  (Environment.SpecialFolder.MyMusic)
Dim Mp3Info As New Mp3.MP3Info
' ListView einrichten
ListView1.Columns.Add("Name", 360)
ListView1.Columns.Add("Dauer", 80)
ListView1.Columns.Add("Bitrate", 80)
ListView1.Columns.Add("Abtastrate", 80)
ListView1.View = View.Details
For Each Fi As FileInfo In New DirectoryInfo(MusikPfad). _
    GetFiles("*.mp3")
  Dim Lv As ListViewItem = ListView1.Items.Add(Fi.Name)
  Mp3Info.Filename = Fi.FullName
  Dim T As New TimeSpan(0, 0, Mp3Info.Length)
  Lv.SubItems.Add(T.ToString)
  Lv.SubItems.Add(Mp3Info.Bitrate)
  Lv.SubItems.Add(Mp3Info.SamplingRateFrequency)
Next
```

Abbildung 9.13: Das ListView-Control in Aktion

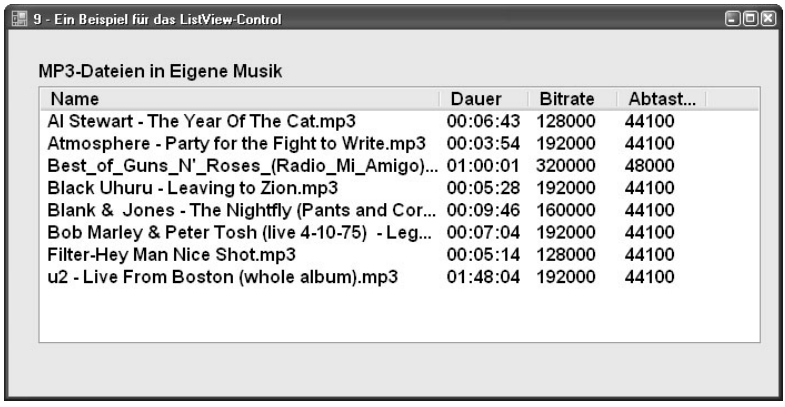


Tabelle 9.6: Interessante Mitglieder der ListView-Klasse

Mitglied	Bedeutung
AllowColumnReorder-Eigenschaft	Wenn *True*, darf der Anwender die Anordnung der Spalten verschieben.
CheckBoxes-Eigenschaft	Wenn *True*, wird neben jedem Eintrag eine CheckBox angezeigt.
CheckedIndices-Eigenschaft	Steht für eine *CheckedIndexCollection*, welche die Indizes aller selektierten Elemente enthält.
CheckedItems-Eigenschaft	Steht für eine *CheckedListViewItemCollection*, die alle selektierten ListView-Elemente enthält.
Clear-Methode	Entfernt alle Elemente und Spalten.
Columns-Eigenschaft	Steht für die Auflistung aller Spalten in Gestalt einer *ColumnHeaderCollection*.
FindItemWithText-Methode	Lokalisiert das erste *ListViewItem*-Objekt mit dem angegebenen Textwert.
FocusedItem-Eigenschaft	Steht für das aktuell selektierte *ListViewItem*-Objekt.
FullRowSelect-Methode	Wenn *True*, wird beim Anklicken eines Eintrags die gesamte Reihe selektiert.
GetItemAt-Methode	Lokalisiert ein *ListView*-Element anhand seiner Position.
HeaderStyle-Eigenschaft	Legt fest, ob der Spaltenkopf anklickbar ist oder nicht.
Items-Eigenschaft	Steht für alle Elemente in Gestalt einer *ListViewItemCollection*.
LabelEdit-Eigenschaft	Wenn *True*, kann der Inhalt eines Elementes (durch erneutes Anklicken) editiert werden.
ListViewItemSorter-Eigenschaft	Erlaubt das Festlegen eines Vergleicherobjekts, das *IComparer* implementiert.

Mitglied	Bedeutung
Scrollable-Eigenschaft	Wenn *True*, wird eine Bildlaufleiste angezeigt.
SelectedItems-Eigenschaft	Steht für die aktuell selektierten Elemente in Gestalt einer *SelectedListViewItemCollection*.
Sort-Methode	Sortiert die Einträge nach der ersten Spalte alphabetisch (dies ist der normalerweise benötigte Fall).
Sorting-Eigenschaft	Legt fest, ob aufsteigend oder absteigend sortiert wird.
View-Eigenschaft	Legt den Anzeigemodus des ListView-Controls fest.

Tabelle 9.6: Interessante Mitglieder der ListView-Klasse (Forts.)

Datum komfortabel auswählen – das MonthCalendar-Steuerelement

Für die komfortable Auswahl eines Datums gibt es gleich zwei Steuerelement: *MonthCalendar* und *DateTimePicker*. Während Ersteres ein relativ attraktives Kalenderblatt darstellt, ist Letzteres eine etwas überdimensionierte TextBox, die ein Datum anzeigt und bei der sich das Kalenderblatt nur auf Mausklick öffnet. Ein wenig irritierend ist, dass es scheinbar keine Eigenschaft gibt, die für das ausgewählte Datum steht. Das »Geheimnis« besteht darin, dass stets ein Datumsbereich markiert werden kann und das selektierte Datum daher über die *SelectionStart*-Eigenschaft abgefragt wird, die für einen *Date*-Wert steht.

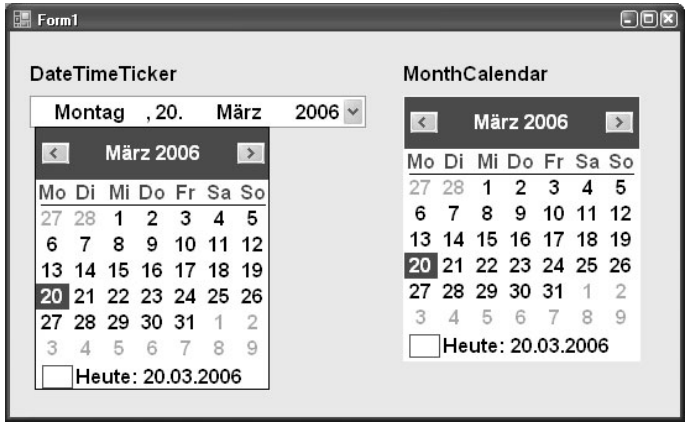


Abbildung 9.14: Die Datumsauswahl in einer Form, wie man sie von Windows gewohnt ist

Mitglied	Bedeutung
BoldedDates-Eigenschaft	*Date*-Array mit Datumswerten, die fett hervorgehoben werden sollen.
MaxSelectionCount-Eigenschaft	Maximale Anzahl an auswählbaren Tagen (maximal 7).
SelectionEnd-Eigenschaft	Das Enddatum eines Datumsbereichs.

Tabelle 9.7: Interessante Mitglieder der MonthCalendar-Klasse

Tabelle 9.7: Interessante Mitglieder der MonthCalendar-Klasse (Forts.)

Mitglied	Bedeutung
SelectionStart-Eigenschaft	Das Startdatum eines Datumsbereichs.
SelectionRange-Eigenschaft	Steht für den ausgewählten Datumsbereich als *SelectionRange*-Objekt.
SetDate-Methode	Legt ein Datum fest, das selektiert werden soll.
SetSelectionRange-Methode	Legt einen Datumsbereich fest, der selektiert werden soll.
ShowToday-Eigenschaft	*True*, wenn das aktuelle Datum hervorgehoben werden soll.
ShowTodayCircle-Eigenschaft	*True*, wenn das aktuelle Datum mit einem Kreis hervorgehoben werden soll.
ShowWeekNumbers-Eigenschaft	*True*, wenn auch die Wochennummer angezeigt werden soll.
TodayDate-Eigenschaft	Steht für das aktuelle Datum des Kalenders (es muss also nicht immer automatisch das aktuelle Datum sein).

Variable Bereiche mit dem Splitter-Steuerelement und dem SplitContainer

Mithilfe des SplitContainers-Controls erhalten Sie eine Fläche, deren Bereiche beliebig verschoben werden können, so wie es zum Beispiel bei einem Ordnerfenster in der Explorer-Ansicht üblich ist. Der Umgang mit dem *SplitContainer*-Control ist vollkommen selbsterklärend. Nachdem das Control auf einem Formular platziert wurde, können auf den beiden Panel-Flächen Steuerelemente angeordnet werden. Lediglich das Feinjustieren der Elemente ist etwas knifflig. Hier hilft es, sich daran zu erinnern, dass jedes Container-Control am einfachsten durch Anklicken mit der rechten Maustaste und Aufrufen des entsprechenden <ELEMENT> AUSWÄHLEN-Eintrags markiert wird.

Tabelle 9.8: Interessante Mitglieder der SplitContainer-Klasse

Mitglied	Bedeutung
IsSplitterFixed-Eigenschaft	Legt fest, ob der Splitter verschiebbar sein soll, bzw. gibt den aktuellen Zustand zurück. *False* bedeutet Splitter ist verschiebbar.
Orientation-Eigenschaft	Legt fest, ob der Splitter horizontal oder vertikal verschoben werden soll.
Panel1-Eigenschaft	Steht für den linken bzw. oberen Bereich in Gestalt eines *SplitterPanel*-Objekts.
Panel1MinSize-Eigenschaft	Mindestgröße des linken bzw. oberen Bereichs.
Panel2-Eigenschaft	Steht für den rechten bzw. unteren Bereich in Gestalt eines *SplitterPanel*-Objekts.
Panel2MinSize-Eigenschaft	Mindestgröße des rechten bzw. unteren Bereichs.

Mitglied	Bedeutung
ProcessTabKey-Eigenschaft	Wählt das nächste Steuerelement aus (über *True* wird die Vorwärts-Richtung eingestellt).
SplitterDistance-Eigenschaft	Position des Splitters in Relation zum linken oder oberen Rand.
SplitterIncrement-Eigenschaft	Legt fest, mit welcher Schrittweite sich der Splitter bewegt.
SplitterWidth-Eigenschaft	Breite des Splitters.

Tabelle 9.8: Interessante Mitglieder der SplitContainer-Klasse (Forts.)

Nur bestimmte Eingaben zulassen – die MaskedTextBox

Eine maskierbare TextBox regelt die Eingabe anhand einer Maske, sodass die Eingabe nur exakt nach einem Muster erfolgen kann, etwa einer internationalen Telefonnummer oder einer Kreditkartennummer. Ein solches Steuerelement gab es bereits vor »Urzeiten« (genauer bei der Version 3.0 von Visual Basic). Da dieses nicht besonders durchdacht war, verschwand es bald wieder. Die neue MaskedTextBox hat mit ihrer Vorgängerin zum Glück nicht mehr viel zu tun. Sie wurde nicht nur funktional erweitert, sondern sie arbeitet tadellos und weist keine Ungereimtheiten auf. Über die *Mask*-Eigenschaft wird eine Eingabemaske ausgewählt oder festgelegt (es wird bereits eine Reihe von Masken angeboten – die einfachste Regel lautet, dass jede Ziffer durch eine »0« repräsentiert wird). Natürlich lässt sich die Box, die sich wie die *TextBox*-Klasse von der *TextBoxBase*-Klasse ableitet, weiter konfigurieren – die wichtigsten Mitglieder der *MaskedTextBox*-Klasse zeigt Tabelle 9.9. Neben den Eigenschaften spielen auch zwei Events eine Rolle: *MaskChanged* und *MaskInputRejected*. Letzteres wird immer dann ausgelöst, wenn eine Eingabe aus irgendeinem Grund zurückgewiesen wurde.

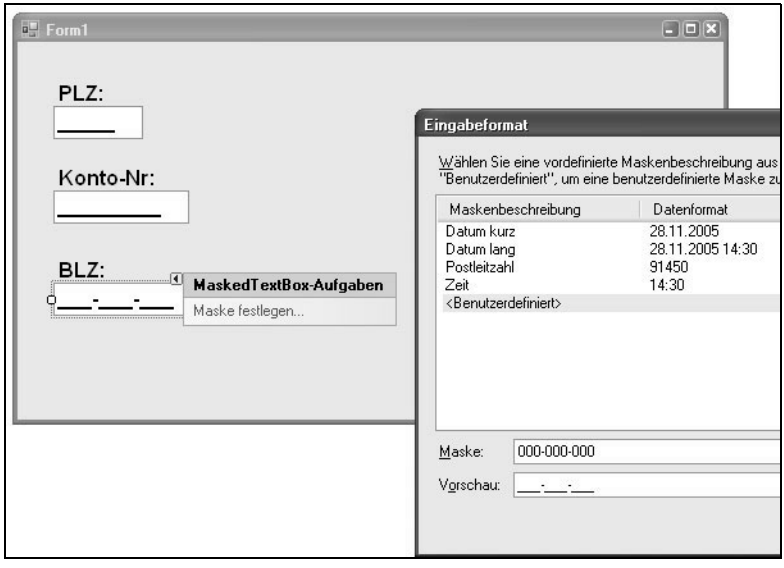


Abbildung 9.15: Das MaskedTextBox-Control in Aktion – es stehen bereits ein paar vordefinierte Masken zur Auswahl.

Sie finden das Beispiel auf der Buch-CD in der Projektdatei 09_MaskedEditBox.sln.

Die Validierung der Eingabe

Eine Maske alleine garantiert noch keine valide (sprich gültige) Eingabe. Ein Betrag kann außerhalb eines erlaubten Bereichs liegen oder eine Telefonnummer in der Form nicht existieren. Ganz ohne Validierung geht es daher nicht. Aber es wird gewährleistet, dass der zu überprüfende Wert grundsätzlich in einem bestimmten Format vorliegt. Sowohl eine TextBox als auch eine MaskedTextBox bieten dafür die Ereignisse *Validating* (Eingabe wird validiert) und *Validated* (Eingabe wurde validiert), die beide von der *Control*-Klasse abstammen. Während beim *Validated*-Event »alles zu spät« ist, kann der *Validating*-Event durch Setzen der *Cancel*-Eigenschaft des *e*-Parameters auf *True*, der vom Typ *CancelEventArgs* ist, abgebrochen werden. Der Fokus verbleibt in diesem Fall in der TextBox.

Bei der MaskedTextBox gibt es zusätzlich den *TypeValidationCompleted*-Event. Er wird ausgelöst, wenn die Typenvalidierung unter Verwendung jenes Typs, der zuvor der *ValidatingType*-Eigenschaft zugewiesen wurde, abgeschlossen wurde. Damit kann erreicht werden, dass der Inhalt der MaskedTextBox einen bestimmten Typ aufweist, zum Beispiel *Short*.

Besitzt die *ValidatingType*-Eigenschaft einen Wert, sieht die Abfolge bei der Validierung bei der MaskedTextBox wie folgt aus:

1. Die MaskedTextBox verliert den Eingabefokus.
2. Die *ValidateText*-Methode wird mit dem Inhalt der *Text*-Eigenschaft aufgerufen.
3. Die durch die *ValidatingType*-Eigenschaft indirekt festgelegte *Parse*-Methode des Typs (zum Beispiel *Integer*) wird aufgerufen, um aus dem Text einen Wert dieses Typs abzuleiten.
4. Unabhängig davon, ob *Parse* den Inhalt erfolgreich in den Typ konvertieren konnte oder nicht, wird der *TypeValidationCompleted*-Event ausgelöst. Die *IsValid*-Eigenschaft des *e*-Parameters gibt an, ob die Umwandlung erfolgreich war.
5. Es wird der *Validating*-Event aufgerufen, der gegebenenfalls durch Setzen von *Cancel* auf *True* abgebrochen wird.
6. Daraufhin wird der *Validated*-Event ausgelöst. Damit wurde die Eingabe vollständig validiert.
7. Es wird der *LostFocus*-Event aufgerufen.

Die Rolle der ErrorProvider

An dieser Stelle wird es Zeit, zumindest mit einem Abstecher auf die *ErrorProvider*-Komponente der Toolbox einzugehen, die sehr nützlich ist, wenn es darum geht, Fehler, die aus einer Eingabevalidierung entstehen, anzuzeigen. Das Konzept der »Windows Forms Provider« ist auf den ersten Blick ein wenig gewöhnungsbe-

dürftig, aber sehr einfach. Der Name rührt von dem Umstand, dass sie anderen Steuerelementen etwas zur Verfügung stellen, in diesem Fall das Anzeigen eines Fehlerzustandes. Wenn Sie einen ErrorProvider auf einem Formular anordnen, passiert noch nicht viel. In Aktion tritt er dann, wenn bei einem Steuerelement ein Fehlerzustand angezeigt werden soll, was zum Beispiel im Rahmen der *Validating*-Ereignisprozedur einer Textbox geschieht.

Das folgende Beispiel »schlägt Alarm«, wenn die Eingabe in der Textbox kleiner als 1 und größer als 49 ist:


```
' Klammern sind wichtig, sonst Fehler
If IsNumeric(tbEingabe1.Text) AndAlso (tbEingabe1.Text < 1 Or ↵
   tbEingabe1.Text > 49) Then
  ErrorProvider1.SetError(tbEingabe1, "Zahl muss zwischen 1 und 49 liegen")
  ElseIf IsNumeric(tbEingabe1.Text) = False Then
    ErrorProvider1.SetError(tbEingabe1, "Eingabe ist keine Zahl")
  Else
    ErrorProvider1.SetError(tbEingabe1, "")
End If
```

Dem ErrorProvider werden zwei Dinge mitgeteilt: der Namen des Steuerelements und der Text der Fehlermeldung. Anschließend erscheint ein blinkendes rotes Ausrufezeichen rechts neben dem Steuerelement. Aufgehoben wird der Fehlerzustand durch einen erneuten Aufruf von *SetError*, dieses Mal mit einem Leerstring.

Sie finden das Beispiel in der Projektdatei 09_ErrorProvider.sln.

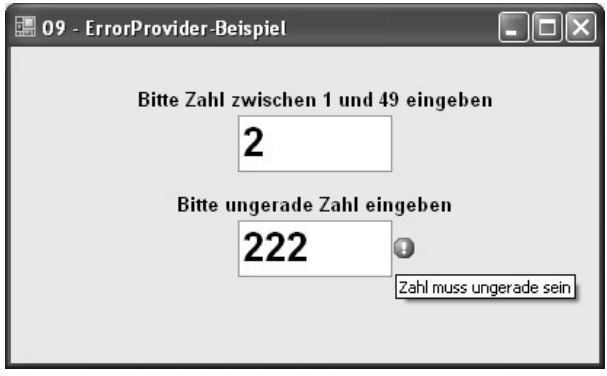


Abbildung 9.16: Der ErrorProvider bietet eine elegante Methode, um Fehler anzuzeigen.

Der Aufbau einer Maske

Der Aufbau einer Eingabemaske ist ausführlich in der Hilfe beschrieben, sodass es hier bei zwei Angaben bleiben soll, die für den Start vollkommen ausreichen: Jede Ziffer wird durch eine »0« repräsentiert. Soll in der Maske kein Promptzeichen, sondern eine 0 erscheinen, muss dieser das Escapezeichen »\« vorausgehen, das generell die Wirkung des folgenden Promptzeichens aufhebt.

Tabelle 9.9: Interessante Mitglieder der MaskedTextBox-Klasse

Mitglied	Bedeutung
AllowPromptAsInput-Eigenschaft	*True*, wenn auch das Promptzeichen – normalerweise der Unterstrich (»_«) – als Eingabe erlaubt ist.
AsciiOnly-Eigenschaft	*True*, wenn nur ASCII-Zeichen im Zeichencodebereich 32 bis 127 erlaubt sind.
BeepOnError-Eigenschaft	*True*, wenn ein zurückgewiesener (da gemäß der Maskendefinition unzulässiger) Tastendruck mit einem Signalton quittiert werden soll.
CutCopyMaskFormat-Eigenschaft	Eigenschaft vom Enumerationstyp *MaskFormat*, die festlegt, wie Promptzeichen bei Cut & Copy-Operationen behandelt werden.
HidePromptOnLeave-Eigenschaft	*True*, wenn die Promptzeichen unsichtbar werden sollen, wenn das Eingabefeld den Fokus verliert.
HideSelection-Eigenschaft	*True*, wenn der markierte Text nicht hervorgehoben bleiben soll, wenn das Steuerelement den Fokus verloren hat.
InsertKeyMode-Eigenschaft	Legt den Einfügemodus als Konstantenauflistung vom Typ *InsertKeyMode* fest.
Mask-Eigenschaft	Legt die Eingabemaske als Zeichenfolge fest (die Hilfe enthält eine Tabelle der erlaubten Zeichen).
RejectInputOnFirstFailure-Eigenschaft	*True*, wenn die Eingabe nach Auftreten des ersten Eingabefehlers nicht weiter untersucht wird.
ResetOnPrompt-Eigenschaft	*True*, wenn die Eingabe eines Promptzeichens dazu führt, dass das Promptzeichen wiederhergestellt wird (Default). *False*, wenn das Promptzeichen als normale Eingabe behandelt wird und der Cursor auf die nächste Position springt.
ResetOnSpace-Eigenschaft	*True*, wenn die Eingabe eines Leerzeichens dazu führt, dass das Promptzeichen wiederhergestellt wird.
ShortcutsEnabled-Eigenschaft	*True*, wenn für die TextBox bestimmte Shortcuts erlaubt sein sollen, zum Beispiel Strg + Z (allgemeine *TextBoxBase*-Eigenschaft).
SkipLiterals-Eigenschaft	*True*, wenn die Eingabe von Maskenzeichen (»Literals«) dazu führt, dass die nächste Eingabeposition angesteuert wird. Bei *False* ist ein Eingabefehler die Folge.
TextMaskFormat-Eigenschaft	*Legt* als Auflistungskonstante vom Typ *MaskFormat* fest, ob die Maske auch Prompt- und Literalzeichen enthalten darf.

TextBox mit Formatierung – die RichTextBox

Das Rich Text Format (kurz RTF) galt zwar schon immer als ziemlich »uncool« und als eine Art »HTML für Arme«, doch es besitzt einen unschätzbaren Vorteil. Es funktioniert und tut genau das, was vom ihm erwartet wird: Texte mit Formatierungsmerkmalen anzureichern (und wenn es sein muss, auch mit eingebetteten Bildern), und das in einem Format, das zum Beispiel von jedem gängigen Textprogramm dargestellt werden kann. Lediglich in einem Webbrowser kann es nicht angezeigt werden. Benötigen Sie daher eine TextBox, die auch eine formatierte Eingabe ermöglicht, ist die *RichTextBox*-Klasse der ideale Kandidat. Da das RTF lediglich eine Option ist, eignet sich die RichTextBox auch als Alternative zur regulären TextBox.

Sie finden das Beispiel auf der Buch-CD in der Projektdatei 09_RTFBox.sln.

Das Geheimnis der Formatierung besteht darin, diese auf den selektierten Text anzuwenden. Netterweise stellt die *RichTextBox*-Klasse mit Eigenschaften wie *SelectionFont* oder *SelectionColor* gleich die passenden Eigenschaften bereit, sodass die Formatierung sehr einfach ist:

```
RichTextBox1.SelectionFont = New Font(RichTextBox1.SelectionFont, _
  FontStyle.Bold)
```

Wenn Sie sich wundern sollten, dass der Text scheinbar nicht fett formatiert dargestellt wird, liegt dies in der Regel lediglich daran, dass er bereits in Fettschrift dargestellt wird und ein vermeintliches Setzen der *Bold*-Eigenschaft dann keine Auswirkung mehr hat. Man muss daher vorher abfragen, ob die *Bold*-Eigenschaft den Wert *True* hat, und dann das *Bold*-Attribut gegebenenfalls auch zurücksetzen. In der *FontStyle*-Enumeration werden die Attribute *Bold*, *Italic*, *Regular*, *Strikeout* und *Underline* zusammengefasst, die per *Or*-, *And*- oder *Xor*-Operator entsprechend kombiniert werden müssen. Der folgende Befehl formatiert den Text fett, ohne die übrigen Attribute zu beeinflussen:

```
RichTextBox1.SelectionFont = New Font(RichTextBox1.SelectionFont, _
  RichTextBox1.SelectionFont.Style Or FontStyle.Bold)
```

Der folgende Befehl entfernt die Fettschrift wieder, ohne die anderen Attribute zu verändern:

```
RichTextBox1.SelectionFont = New Font(RichTextBox1.SelectionFont, _
  RichTextBox1.SelectionFont.Style Xor FontStyle.Bold)
```

Der kleine Unterschied ist ein »X«, genauer gesagt der *Xor*-Operator, der anstelle von *Or* zum Einsatz kommt.

Abbildung 9.17: Ein formatierter Text einmal in der RichTextBox ...

Abbildung 9.18: ... und einmal in einer bekannten Textverarbeitung

Tabelle 9.10: Interessante Mitglieder der RichTextBox-Klasse

Mitglied	Bedeutung
AcceptTabs-Eigenschaft	*True*, wenn das Drücken der Tab-Taste nicht dazu führen soll, dass das nächste Steuerelement angesprungen wird.
AppendText-Methode	Hängt einen Textblock an das Ende der RichText-Box an.
AutoWordSelection-Eigenschaft	*True*, wenn ein Wort durch einen Doppelklick markiert wird (einen Unterschied zur *False*-Einstellung konnte der Autor allerdings nicht feststellen).
CanUndo-Eigenschaft	*True*, wenn die letzte Operation rückgängig gemacht werden kann.
Clear-Methode	Löscht den Inhalt der RichTextBox.
DeselectAll-Methode	Hebt die Auswahl wieder auf.

Tabelle 9.10: Interessante Mitglieder der RichTextBox-Klasse (Forts.)

Mitglied	Bedeutung
DetectUrls-Eigenschaft	*True*, wenn eine URL speziell formatiert werden soll.
DrawToBitmap-Methode	Rendert den Inhalt der RichTextBox in eine externe Bitmap. Diese mit dem .NET Framework 2.0 eingeführte Methode erbt die Klasse von der allgemeinen *Control*-Klasse. Beim Rendern werden aber (offenbar) in der Innenfläche angeordnete Steuerelemente und nicht der Text berücksichtigt.
GetCharFormPosition-Methode	Liefert das Zeichen als *Char*-Wert, das dem übergebenen Punkt am nächsten ist.
GetFirstCharIndexFromLine-Methode	Liefert die Position des ersten Zeichens in der angegebenen Zeile. Auf diese Weise lässt sich eine Zeile heraustrennen.
LoadFile-Methode	Lädt eine Textdatei im RTF- oder ASCII-Format.
RightMargin-Eigenschaft	Liefert oder setzt den rechten Rand des Eingabefeldes.
Rtf-Eigenschaft	Der komplette Inhalt der RichTextBox im RTF-Format.
SaveFile-Methode	Speichert den Inhalt der RichTextBox in eine Datei oder einen Stream.
SelectedText-Eigenschaft	Der markierte Text.
Text-Eigenschaft	Der komplette Inhalt der RichTextBox im ASCII-Format.
Undo-Methode	Macht die letzte Änderung rückgängig.

HTML-Dokumente anzeigen – das WebBrowser-Steuerelement

Ein Webbrowser ist im Jahre 2006 wahrlich nichts Sensationelles mehr. Auch das Platzieren eines Webbrowsers auf einem Formular in Gestalt eines ActiveX-Controls, sodass dort eine Webseite angezeigt werden konnte, ließ sich im Prinzip bereits bei Visual Basic 4.0 realisieren. Das neue WebBrowser-Control ist dennoch etwas Besonderes, denn es handelt sich nicht mehr um ein ActiveX-Control, das lediglich in einem »Korsett« auf ein Windows-Formular gehievt wird. Die Browser-Engine wurde vielmehr in eine Klasse eingehüllt, die sich von der *Control*-Klasse ableitet und damit nicht nur einen konsistenten Satz an Eigenschaften und Methoden anbietet, sondern auch den üblichen Komfort eines Steuerelements aufweist wie eine *DrawToBitmap*-Methode, mit der sich der Inhalt des angezeigten HTML-Dokuments in eine Bitmap umwandeln lässt.

Das folgende Beispiel lädt eine Webseite in das WebBrowser-Control:

```
Dim Url As New Uri("http://www.mut.de")
wb.Navigate(Url)
```

Abbildung 9.19: Das WebBrowser-Control zeigt den Inhalt der geladenen Webseite an.

Zugriff auf das DOM

Ist ein HTML-Dokument erst einmal im Kasten, sprich im Arbeitsspeicher des Programms, möchte man vielleicht auch auf dessen Inhalt zugreifen. Hier kommt das *Document Object Model* (DOM) ins Spiel, das jedes HTML-Dokument als Objektbaum repräsentiert, dessen Objekte mit den Seitenelementen korrespondieren und dessen Struktur der stets hierarchischen Struktur einer HTML-Seite entspricht. Damit lassen sich zum Beispiel alle Tags, alle Frames, alle Skriptblöcke und deren Inhalte nicht nur ansprechen und enumerieren, sondern auch verändern.

Der folgende Codeschnipsel listet im DocumentCompleted*-Ereignis die Namen aller Tags des geladenen Dokuments auf:*

```
Dim Wb As WebBrowser = CType(sender, WebBrowser)
lbDOM.Items.Clear()
For Each T As HtmlElement In Wb.Document.All
    If T.TagName <> "" Then lbDOM.Items.Add(T.TagName)
Next
```

Die Möglichkeit, HTML-Dokumente auch editieren zu können, bietet das WebBrowser-Control nicht. Wer das möchte, muss nach wie vor auf die (inzwischen recht betagte) COM-Komponente DHTML Editor Control *zurückgreifen, die Bestandteil des Internet Explorers ist und daher auf jedem System vorhanden sein sollte (einen anschaulichen Artikel, der dieses im Grunde recht interessante Control in Aktion zeigt, finden Sie unter* http://www.microsoft.com/germany/msdn/library/web/DasDHTMLEditorControlMitVBNETVerwendet.mspx).

Den Webbrowserinhalt als Bitmap speichern

Eine einfache »Screenshot«-Funktion lässt sich durch den Umstand realisieren, dass sich auch die *WebBrowser*-Klasse als echte Steuerelementklasse von der *Control*-Klasse ableitet und diese seit dem .NET Framework 2.0 eine *DrawToBitmap*-Methode anbietet, mit der sich der Inhalt eines Steuerelements in eine Bitmap »rendern« lässt (so wird das Umwandeln in etwas Sichtbares allgemein bezeichnet).

Das folgende Beispiel »schießt« ein Bildschirmfoto von dem Inhalt des WebBrowser-Control-Fensters, speichert dieses als Bitmap und zeigt es anschließend über die Start*-Methode der* Process*-Klasse mit dem Programm Paint an:*

```
Dim Bm As New Bitmap(wb.ClientRectangle.Width, _
  wb.ClientRectangle.Height)
wb.DrawToBitmap(Bm, wb.ClientRectangle)
' Nicht zwingend erforderlich
wb.BringToFront()
Bm.Save("C:\WbBitmap.bmp")
Process.Start("MsPaint", "C:\WbBitmap.bmp")
```

Anzeigen von animierten Gifs

Im Zusammenhang mit dem letzten Beispiel kommt ein weiterer Aspekt ins Spiel, der sich unter VB6 nicht ohne Hilfsmittel lösen ließ: das Anzeigen und vor allem Steuern von animierten Gif-Dateien. Die Aufgabenstellung könnte etwas folgendermaßen aussehen: Während ein HTML-Dokument geladen wird, soll ein rotierender Globus dargestellt werden, der aus einer simplen animierten Gif-Datei (*World.gif*) mit einer Reihe von Bitmaps besteht, die nacheinander erscheinen. Das Anzeigen eines animierten Gifs ist grundsätzlich kein Problem. Es wird einfach ein *Bitmap*-Objekt instanziiert und der *Image*-Eigenschaft einer PictureBox zugewiesen:

```
Dim Bm As New Bitmap("World.gif”)
pbStatus.Image = Bm
```

Um das animierte Gif auch steuern zu können, bedarf es der *ImageAnimator*-Klasse und ihrer freigegebenen Methoden *Animate* und *StopAnimate*:

```
ImageAnimator.Animate(pbStatus.Image, FrameEv)
```

In beiden Fällen muss ein simpler Eventhandler übergeben werden, der eine Ereignisprozedur definiert, die aufgerufen wird, wenn die Animation läuft:

```
Private FrameEv As New EventHandler(AddressOf FrameChange)
```

Auch wenn die Eventprozedur (pro Frame) aufgerufen wird, passiert hier nicht viel, zumal aufgrund des allgemeinen Typs des Eventhandlers zum Beispiel nicht die Framenummer übergeben wird.

Tabelle 9.11: Interessante Mitglieder der Web-Browser-Klasse

Mitglied	Bedeutung
AllowNavigation-Eigenschaft	*True*, wenn das Navigieren innerhalb der bereits angezeigten Seiten erlaubt sein soll.
Document-Eigenschaft	Steht für das HTML-Objektmodell, das jedes Seitenelement durch ein entsprechendes Objekt repräsentiert.
GoBack-Methode	Ruft die zuletzt angezeigte Seite auf.
Print-Methode	Druckt den Seiteninhalt aus.
ScriptErrorsSuppressed-Eigenschaft	*True*, wenn Skriptfehler beim Laden eines Dokuments nicht angezeigt werden sollen (die Standardeinstellung ist *False*).
StatusText-Eigenschaft	Steht für den aktuellen Zustand des Web-Browsers als Zeichenkette.

9.4 Ziehen-und-Ablegen-Operationen

Bei einer Ziehen-und-Ablegen-Operation wird ein Steuerelement bei gedrückter Maustaste an einen anderen Ort innerhalb desselben oder eines anderen Formulars gezogen und dort abgelegt. Sie sind bei WinForms-Formularen und ihren Steuerelementen fest eingebaut. Gegenüber der »neumodischen« Variante des Ziehens und Ablegens bei VB6 gibt es nur weniger Unterschiede.

Die Drag & Drop-Implementierung bei den Windows Forms-Formularen ist dem OLE-Drag & Drop früherer Visual Basic-Versionen sehr ähnlich.

Ziehen und Ablegen starten

Eine Ziehen-und-Ablegen-Operation ist mit jedem Steuerelement möglich, sie muss aber explizit gestartet werden. Dies geschieht über die *DoDragDrop*-Methode des Steuerelements, die am besten im *MouseDown*-Ereignis aufgerufen wird.

Das folgende Beispiel startet eine Ziehen-und-Ablegen-Operation mit der ListBox liSpieler*, wenn die linke Maustaste gedrückt wird:*

```
If e.Button = Windows.Forms.MouseButtons.Left Then
  liSpieler.DoDragDrop(liSpieler.SelectedItem, DragDropEffects.Copy)
End If
```

Der *DoDragDrop*-Methode werden zwei Argumente übergeben:

- das zu ziehende Element (in diesem Fall der aktuell selektierte Listeneintrag, bei dem es sich auch um ein Objekt handeln kann),
- die durchzuführende Operation, die über die Enumeration *DragDropEffects* ausgewählt wird.

Im obigen Fall wurde *Copy* gewählt. Dies betrifft in erster Linie das visuelle Feedback (je nachdem, ob *Copy* oder *Move* verwendet wird, wird ein anderes Symbol angezeigt), hat aber keinen Einfluss auf die tatsächlich durchgeführte Operation, denn diese wird vom Programmierer festgelegt. Die Auswahl dient also lediglich dazu, den (optischen) Modus der Operation festzulegen, sodass der Mauszeiger die entsprechende Form erhält. Was im Einzelnen passiert, spielt keine Rolle.

Auf eine Ablegen-Operation reagieren

Damit ein Steuerelement der Empfänger einer Ablegen-Operation werden kann, muss dessen *AllowDrop*-Eigenschaft auf *True* gesetzt werden. Ist dies geschehen, wird eine Reihe von Ereignissen ausgelöst, sobald sich ein Objekt im »Hoheitsgebiet« (also in der sichtbaren Fläche) des Steuerelements befindet (siehe Tabelle 9.12). Die wichtigsten Ereignisse sind *DragEnter* und *DragDrop*. Bei *DragEnter* betritt das Steuerelement den Bereich des potenziellen Ziels, und dieses muss gegebenenfalls ein visuelles Feedback geben, indem es das Symbol des Mauszeigers ändert. Bei *DragDrop* wird das Steuerelement abgelegt, und das Zielobjekt muss darauf nun irgendwie reagieren.

Bei beiden Ereignissen ist der *e*-Parameter vom Typ *System.Windows.Forms DragEventArgs*. Dies ist der Schlüssel zu dem, was abgelegt wurde (siehe Tabelle 9.13). Es kann eine Bitmap, ein Text oder ein beliebiges Objekt sein. Das macht das Ziehen und Ablegen sehr leistungsfähig. An die Daten kommt man stets über die *GetData*-Methode heran.

Das folgende Beispiel ist der Inhalt einer DragEnter*-Ereignisprozedur. Sie prüft, ob es in einem Panel* paDreamTeam *bereits ein Label gibt, dessen* Text*-Eigenschaft dem Wert der Eigenschaft* SpielerName *des gezogenen* Spieler*-Objekts entspricht. Sollte dies der Fall sein, wird die Aktion abgebrochen und ein späteres Ablegen verhindert:*


```
If e.AllowedEffect = DragDropEffects.Copy Then
  Dim Sp As Spieler = e.Data.GetData(GetType(Spieler))
  ' Feststellen, ob der Spieler bereits gedroppt wurde
  For Each C As Control In paDreamTeam.Controls
      If TypeOf C Is Label Then
          If Sp.Spielername = CType(C, Label).Text Then
              e.Effect = DragDropEffects.None
              Exit Sub
          End If
      End If
  Next
  e.Effect = DragDropEffects.Copy
End If
```

Diese Prozedur enthält zwei wichtige Programmiertechniken:

- Über *GetData* wird gezielt auf den Typ *Spieler* zugegriffen, der durch *DoDragDrop* übergeben wurde. Der Typ, also das *Type*-Objekt, das hinter der Klasse *Spieler* steht (alles Weitere im nächsten Beispielprojekt), wird über die *GetType*-Methode abgefragt.
- In einer *For Each*-Schleife werden alle Labels durchlaufen und ihre *Text*-Eigenschaft geprüft. Bei einer Übereinstimmung wird durch Setzen der *Effect*-Eigenschaft erreicht, dass ein »Ablegen nicht möglich«-Symbol eingeblendet wird. Der Anwender erkennt daran, dass er das Objekt (aus welchen Gründen auch immer) nicht auf dem Panel ablegen kann. Ist der Name dagegen noch nicht vorhanden, wird das *Copy*-Symbol beibehalten, und der geschulte Anwender erkennt, dass eine Kopie des Objekts abgelegt wird.

Tabelle 9.12: Die Ziehen-und-Ablegen-Ereignisse in der Übersicht

Ereignis	Auslösende Aktion
DragDrop	Das gezogene Element wird auf dem Steuerelement abgelegt.
DragEnter	Das gezogene Element betritt den sichtbaren Bereich des Steuerelements.
DragLeave	Das gezogene Element verlässt den sichtbaren Bereich des Steuerelements wieder.
DragOver	Das gezogene Element bewegt sich im Bereich des Steuerelements.

Tabelle 9.13: Die Elemente der DragEventArgs-Klasse

Element	Bedeutung
AllowedEffect-Eigenschaft	Nur-lesen-Eigenschaft, die alle für das Objekt erlaubten und beim Aufruf von *DoDragDrop* festgelegten Effekte enthält.
Data-Eigenschaft	Steht für ein *DataObject*-Objekt, das einen Zugriff auf die übergebenen Daten erlaubt. Dieses Objekt umfasst die Methoden *GetData*, *GetDataPresent*, *GetFormats* und *SetData*.
Effect-Eigenschaft	Legt fest, welches Symbol der Mauszeiger im weiteren Verlauf der Operation annehmen soll.
KeyState-Eigenschaft	Steht für den aktuellen Zustand der Tasten ⇧, Strg und Alt sowie der Maustasten.
X-Eigenschaft	x-Position des Mauszeigers.
Y-Eigenschaft	y-Position des Mauszeigers.

Sie finden das Beispiel auf der Buch-CD in der Projektdatei 09_PanelDragDrop.sln.

Das folgende Beispiel stellt ein vollständiges Beispiel für eine Ziehen-und-Ablegen-Operation dar. Es besteht aus einem WinForms-Formular, auf dem sich eine ListBox und ein Panel als Spielplan befinden: *liSpieler* und *paDreamTeam*. Fußballfans wissen sicherlich, worum es geht. Für alle anderen: Die Liste enthält eine (willkürliche) Auswahl von Topspielernamen, die auf das Panel mit der Maus gezogen und dort nicht nur abgelegt, sondern auch verschoben werden können (warum nicht, denn das Drag & Drop funktioniert auch innerhalb eines Steuerelements). Da dabei ein *Spieler*-Objekt gezogen wird, kann die Bitmap des Landes, für das der Spieler antritt, angezeigt werden (besser wäre natürlich ein Foto des Spielers, doch dann gibt es unter Umständen Ärger mit den Sponsoren[2]).

Das Listing wird im Folgenden nicht komplett abgebildet, sondern nur die für die Ziehen-und-Ablegen-Operationen wichtigen Bereiche.

Der entscheidende Part ist, dass sich auf einem Panel insgesamt elf *PictureBox*-Controls befinden. In jedem soll das Ablegen eines *Spieler*-Objekts möglich sein. Bei VB6 hätte man diese PictureBoxen zu einem Steuerelementfeld zusammenfassen müssen. Bei Visual Basic 2005 kann dies nachträglich geschehen:

```
' Alle PictureBoxen des Panels zu einem Feld zusammenfassen
For Each C As Control In paDreamTeam.Controls
  If TypeOf C Is PictureBox Then
    PictureFeld(Nr) = C
    With CType(C, PictureBox)
      .AllowDrop = True
      .SizeMode = PictureBoxSizeMode.StretchImage
    End With
    Nr += 1
    AddHandler CType(C, PictureBox).DragDrop, _
      AddressOf Picbox_DragDrop
    AddHandler CType(C, PictureBox).DragEnter, _
      AddressOf Picbox_DragEnter
  End If
Next
```

Alle PictureBoxen teilen sich denselben *DragDrop*-Handler, in dem Folgendes passiert:

- Der *Image*-Eigenschaft wird der Wert der *Flagge*-Eigenschaft des *Spieler*-Objekts zugewiesen, die für eine zuvor zugewiesene Flaggen-Bitmap steht.
- Über den Namen der PictureBox wird das dazugehörige Label lokalisiert und dort der Name des Spielers eingetragen (es wird davon ausgegangen, dass, wenn eine PictureBox zum Beispiel »PictureBox1« heißt, das dazugehörige Label den Namen »Label1« trägt).

2 Das ist natürlich ein wenig ironisch gemeint, wobei man bei der regelwütigen FIFA nie ganz sicher sein kann. Aber generell muss man die Persönlichkeitsrechte bei prominenten Menschen beachten.

```
Sub Picbox_DragDrop(ByVal sender As Object, ByVal e As System. _
    Windows.Forms.DragEventArgs)
  If e.Effect = DragDropEffects.Copy Then
    Dim Sp As Spieler = e.Data.GetData(GetType(Spieler))
    CType(sender, PictureBox).Image = Sp.Flagge
    Dim PicName As String = CType(sender, PictureBox).Name
    ' Nr extrahieren
    Dim Nr As String
    For i As Short = 0 To PicName.Length - 1
      Dim C As Char = PicName.Chars(i)
      If Char.IsNumber(C) Then
        Nr = PicName.Substring(i)
      End If
    Next
    CType(paDreamTeam.Controls.Find("Label" & Nr, False)(0), Label). _
      Text = Sp.Spielername
  End If
End Sub
```

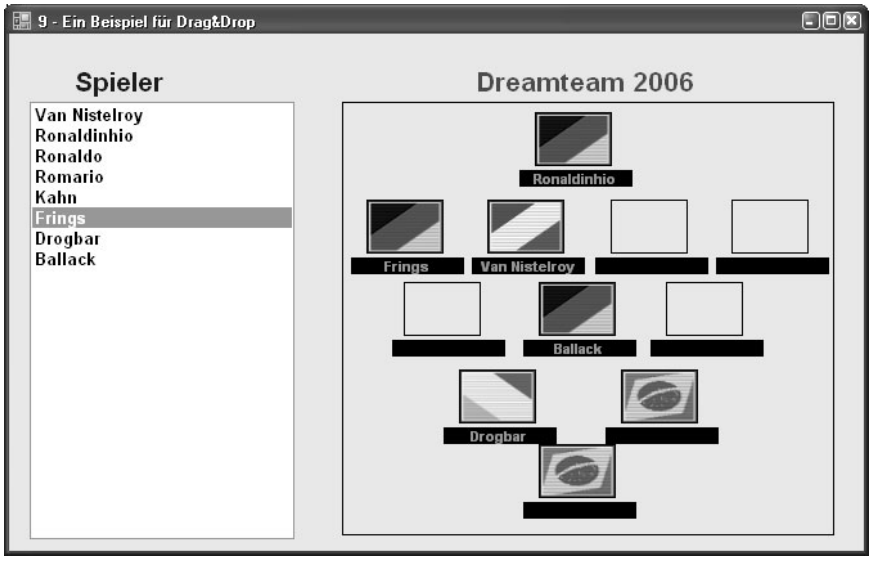


Abbildung 9.20: Wer ist das Dreamteam 2006? Dank Drag & Drop können Sie sich einfach und elegant eine eigene Auswahl zusammenstellen.

9.5 Layout ist mehr als nur das Anordnen von Steuerelementen

Hinter einer Benutzeroberfläche, die alle wichtigen Anforderungen bezüglich Ergonomie, Bedienbarkeit und vor allem Konsistenz erfüllt, steckt sehr viel Arbeit. Arbeit, die der normale Anwender in der Regel nicht sieht, aber als selbstverständlich voraussetzt[3]. Auch Visual Basic 2005 zaubert keine perfekte Oberfläche auf Knopfdruck, bietet aber ein paar Hilfestellungen, die es bei früheren Versionen nicht gab:

- Die Führungslinien, die beim Anordnen eines Steuerelements sicherstellen, dass es bündig zu anderen Steuerelementen angeordnet wird. Diese Neuerung ist einfach und genial.
- Einen »Griff«, der beim Selektieren eines Steuerelements in Gestalt eines kleinen Vierfachpfeils sichtbar wird und über den sich das Steuerelement verschieben lässt (Sie müssen es also nicht mehr irgendwo zu »fassen« kriegen).
- Ein TableLayout-Control, mit dem sich die Steuerelemente anhand einer unsichtbaren Tabelle wie auf einer HTML-Seite anordnen lassen.
- Eine Undo-Funktion, die auch das Verschieben von Steuerelementen rückgängig macht (probieren Sie es einmal aus).
- Verschiedene Verbesserungen, die dafür sorgen, dass sich der Inhalt eines Formulars variabel an die Größe des Formulars anpasst.
- Eine neue Layout-Engine, die für eine modernere Optik sorgt (das typische, altmodische »Visual Basic Look & Feel«, an dem man früher treffsicher eine Visual Basic-Anwendung erkennen konnte, gibt es nicht mehr).
- Durch den Umstand, dass ein Formular vollständig auf Quelltext basiert, ist es für ein Add-In problemlos möglich, ein Formular nachzubearbeiten, indem zum Beispiel die Größe oder auch die Position eines Steuerelements anhand einer Vorgabe überprüft und angepasst wird.

Trotz Führungslinien ist das Anordnen von Steuerelementen oft noch ein Geduldspiel. Am einfachsten ist es daher, die Position und Größe in den zuständigen Eigenschaften *Location* und *Size* im Eigenschaftenfenster auf den Pixel genau einzustellen.

Steuerelemente verankert – die Anchor-Eigenschaft

Über die *Anchor*-Eigenschaft, die jedes Steuerelement besitzt, wird es möglich, dieses mit einer oder mehreren der vier umgebenden »Seitenwände« zu verankern, sodass es diesen Abstand beim Vergrößern und Verkleinern des Formulars beibehält, was wiederum dazu führt, dass es seine Größe anpasst[4]. Verankern bedeutet aber nicht andocken, das heißt, das Steuerelement ändert seine Position nur relativ.

Die *Anchor*-Eigenschaft, die am einfachsten über das Eigenschaftenfenster eingestellt wird, kann insgesamt fünf Einstellungen annehmen: *Bottom*, *Left*, *None*, *Right* und *Top*. Da ein Steuerelement mit mehreren Rändern verankert sein kann, werden die Einstellungen per *Or*-Operator verknüpft.

3 Die Erwartungen sind auch in diesem Bereich grenzenlos. Und wer muss sie im Allgemeinen erfüllen? Der »arme« Entwickler.

4 Dies ist praktisch der lang erwartete Ersatz (es gibt ihn bereits seit Visual Studio .NET 2002) für jene »Resizer-Controls«, die bereits zu Zeiten von Visual Basic 3 populär waren.

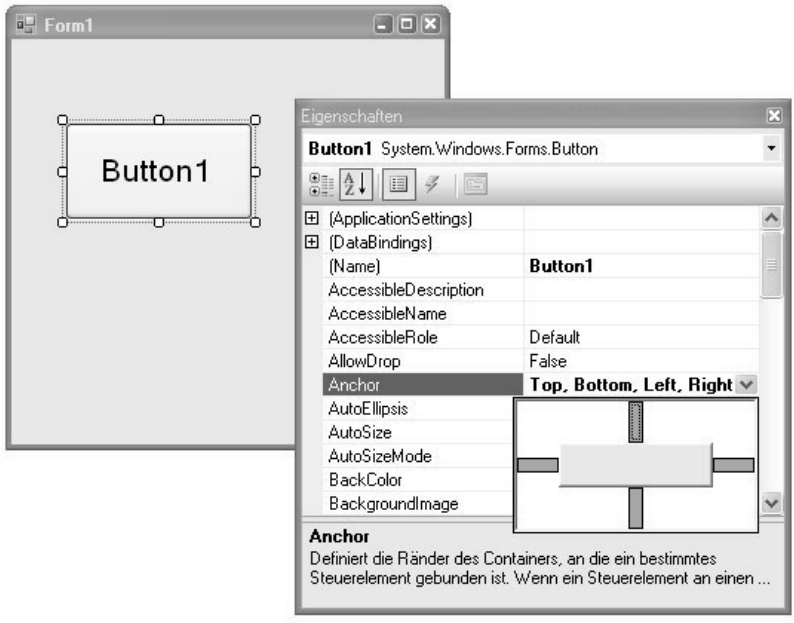


Abbildung 9.21: Die Anchor-Eigenschaft sorgt dafür, dass der relative Abstand zu den ausgewählten Rändern konstant bleibt.

Steuerelemente andocken – die Docking-Eigenschaft

Das Andocken eines Steuerelements ist eine Eigenschaft, die bei früheren Visual Basic-Versionen fest eingebaut war. Einige »Spezialelemente« wie die Tool- oder Statusbar besaßen die Fähigkeit, automatisch an die obere oder untere Umrandung zu »hüpfen«. Bei .NET weist jedes Steuerelement eine *Docking*-Eigenschaft auf, durch die es an einem oder allen der vier Seitenränder andocken kann. Ein Steuerelement, das an alle Seiten andockt, füllt die Innenfläche des Formulars nicht nur komplett aus, sondern ändert seine Größe auch entsprechend. Der SplitContainer ist ein Beispiel für ein Steuerelement, bei dem dieses Verhalten sehr gut passt. Das Andockverhalten wird über die *Dock*-Eigenschaft eingestellt, die insgesamt sechs Einstellungen kennt: *Bottom*, *Fill* (alle Ränder werden angedockt), *Left*, *None*, *Right* und *Top*. Auch hier gilt, dass es am einfachsten ist, die Einstellungen im Eigenschaftenfenster auszuwählen.

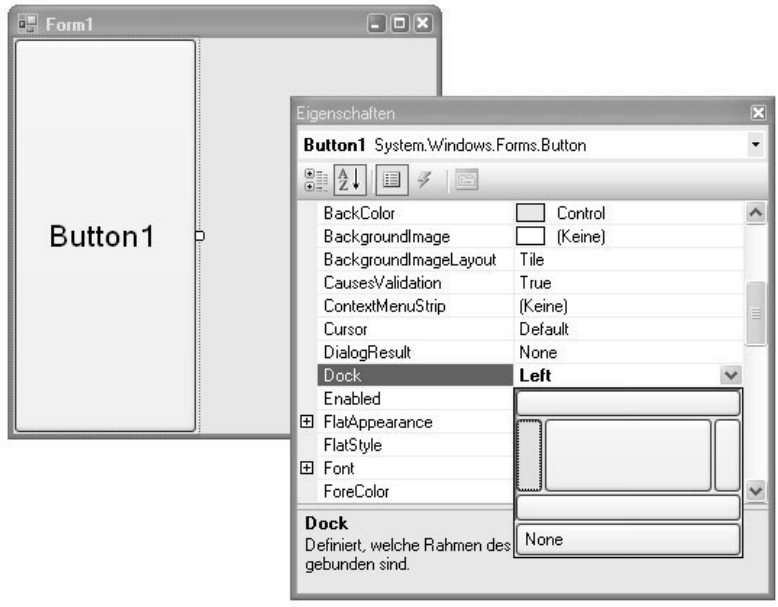


Abbildung 9.22: Über die Dock-Eigenschaft wird ein Andocken des Steuerelements an einem oder allen Rändern erreicht.

Automatische Größenanpassung

Eine automatische Größenanpassung, die durch Setzen einer »AutoSize«-Eigenschaft auf *True* erreicht werden könnte, gibt es auch bei Visual Basic 2005 nicht (und damit keinen direkten Ersatz für die Resizer-Controls früherer Versionen). Wem eine solche automatische Anpassung wichtig ist, erreicht diese über verschiedene Maßnahmen:

- Über das Verankern eines Steuerelements an allen vier Rändern – damit passt sich aber die Schriftgröße nicht automatisch an.
- Über den *Resize*-Event, der bei der Größenänderung eines Steuerelements eintritt. Da es problemlos möglich ist, sämtliche *Resize*-Events aller Steuerelemente (über eine kleine Schleife, welche die *Controls*-Auflistung rekursiv durchläuft) auf einen Handler zu vereinigen, besteht die Möglichkeit, eine zentrale »Anpassungsroutine« einzurichten.

Die AutoSize-Eigenschaft

Es gibt sie doch bei .NET 2.0 – eine *AutoSize*-Eigenschaft. Allerdings nicht für das gesamte Formular, sondern nur pro Steuerelement und mit der Wirkung, dass sich die Größe des Steuerelements an seinen Inhalt (etwa bei einem Label), aber nicht an die Größe des umgebenden Containers anpasst.

Layout in der Tabelle – das TableLayoutPanel-Control

Wer sich bereits ein wenig mit HTML und der Umsetzung kleiner Webanwendungen beschäftigt hat, weiß wahrscheinlich, dass der <Table>-Tag und die damit einhergehenden Tabellen eine beliebte, wenn auch bei Layoutpuristen sehr umstrittene Möglichkeit sind, ein »Layout« zu gestalten[5]. Die Idee ist simpel: Eine unsichtbare Tabelle gibt mit ihren Zeilen und Spalten Positionen vor, an denen Seitenelemente positioniert werden. Da eine pixelgenaue Positionierung bei HTML ansonsten sehr viel Erfahrung voraussetzt und bei alten Browsern nicht funktioniert, sind Tabellenlayouts nach wie vor sehr beliebt.

Mit .NET 2.0 bietet das *TableLayoutPanel*-Control einen ähnlichen Komfort, wenngleich mit HTML absolut keine Gemeinsamkeiten zu verzeichnen sind. Die Idee ist auch hier, dass eine zur Ausführungszeit unsichtbare Tabelle das Anordnen von Steuerelementen auf einem Formular erleichtern soll. Auch wenn das alles nichts mit HTML und dem Web zu tun hat, liegt es nahe, dass sich ein solches Formular vergleichsweise einfach auf eine HTML- oder besser eine XAML-Oberfläche übertragen lässt. XAML steht für *Extended Application Markup Language* und ist bei Microsoft die Zukunft des Oberflächendesigns für Windows- wie Webanwendungen gleichermaßen. Mit der nächsten Version von Visual Studio wird es einen integrierten XAML-Designer geben, sodass wir alle in ein paar Jahren nur noch mit XAML programmieren. Dann wird auch die automatische Größenanpassung inklusive sein. Aber das ist natürlich ein anderes Thema[6].

9.6 Visual Styles bei Windows XP und das Windows-Anwendungsframework

Auf eine »saubere« Unterstützung der mit Windows XP eingeführten »Visual Styles« mussten Programmierer etwas länger warten[7], was insofern schade ist, da es inzwischen Dutzende, wenn nicht Hunderte von Styles gibt. Ob es sinnvoll ist, seinen Computer mit einer solchen »Performancebremse« auszustatten, sei einmal dahingestellt, eine Anwendung sollte aber in der Lage sein, sich automatisch an das aktuelle Design anzupassen. Unter Visual Studio .NET 2003 musste noch vor dem Aufruf von *InitializeComponent* in der Formularklasse der Aufruf von *Application.EnableVisualStyles()* erfolgen. Wurde bei einem Steuerelement, wie einem Button, für die *FlatStyle*-Eigenschaft *System* eingestellt, übernahm ein Steuerelement den vorgegebenen Style (bei den meisten Steuerelementen war es nicht erforderlich). Unter Visual Studio 2005 ist dieser Zusatzschritt nicht mehr erforderlich, die Steuerelemente passen sich (bereits zur Entwurfszeit) automatisch an. Ist dies aus irgendeinem Grund nicht erwünscht, muss lediglich die Option VISUAL XP-STILE AKTIVIEREN in den Anwendungseigenschaften ausgeschaltet werden.

5 Solche Spaßverderber gibt es leider überall.

6 Wenn Sie möchten, können Sie sich eine Vorabversion von »Cider«, so der Name dieses Designers, für Visual Studio 2005 bereits heute ansehen. Oder Sie lesen diese Zeilen erst zu einem Zeitpunkt, zu dem Visual Studio 2007 schon offiziell ist, und wundern sich vielleicht, dass es noch etwas anderes als XAML geben könnte.

7 Immerhin gibt es Windows XP bereits seit Oktober 2001.

Hinter den Styles steckt eine altbekannte Komponente: Comctl32.dll *(welche die »Windows Common Controls« enthält), die inzwischen in der Version 6 vorliegt (und nur unter Windows XP und Windows Server 2003 läuft). Unter Windows Server 2003 muss gegebenenfalls der Systemdienst* Themes *aktiviert und gestartet werden.*

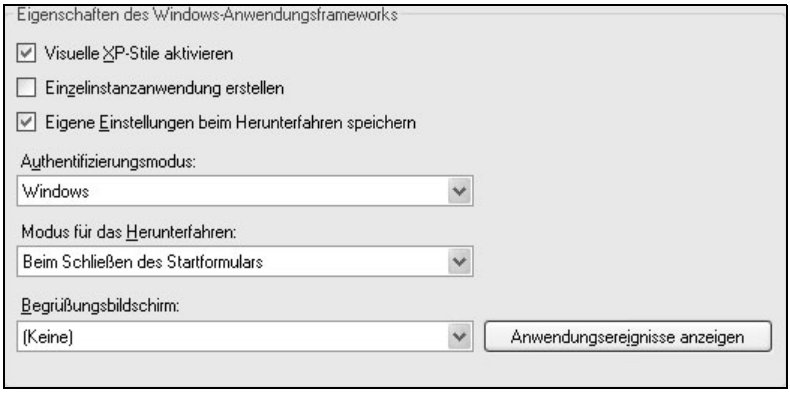


Abbildung 9.23: Zu den Anwendungseigenschaften gehört auch die Option zum Aktivieren der XP-Stile.

9.7 Es gibt sie doch – die Steuerelementfelder

Dieser Abschnitt richtet sich an jene VB6-Programmierer, die immer noch ein wenig ihren Steuerelementfeldern nachtrauern, die es bei .NET nicht mehr gibt. Sie können innerhalb des Windows Forms-Designers Steuerelemente beliebig kopieren und einfügen, ohne dass dadurch ein Steuerelementfeld entsteht. Das heißt aber nicht, dass Steuerelemente nicht mehr zu einem Feld zusammengefasst werden können. Diese Möglichkeit existiert weiterhin, und sie ist deutlich flexibler als früher. Die entsprechende Funktion ist nur nicht mehr zur Entwurfszeit verfügbar, sondern ausschließlich zur Ausführungszeit.

Da Steuerelemente reguläre Klassen sind, kann jedes Steuerelement zur Laufzeit durch Instanziieren der dazugehörigen Klasse aus dem »Nichts« heraus entstehen:

```
Dim bnNeu As New Button()
```

Damit steht *bnNeu* für einen Button. Wird dieser zur *Controls*-Auflistung eines Containers hinzugefügt, wird der Button tatsächlich angezeigt, wenngleich man in der Praxis Größe, Position und andere Attribute setzen würde:

```
Me.Controls.Add(bnNeu)
```

Benötigt man nicht einen, sondern mehrere Buttons, fasst man die Befehle einfach in einer Schleife zusammen und fügt die resultierenden Instanzen in ein Array oder in eine Auflistung ein:

```
Dim ButtonListe As New List(Of Button)

For i As Short = 1 To 26
  bnNeu = New Button()
  ButtonListe.Add (bnNeu)
  Me.Controls.Add (bnNeu)
Next
```

Am Ende enthält die Auflistung *ButtonListe* 26 Buttons. Zwei wichtige Anforderungen fehlen aber noch: Die Buttons müssen unterscheidbar sein, und sie müssen auf Ereignisse reagieren können. Den ersten Punkt erreicht man über die universelle *Tag*-Eigenschaft, den zweiten über den *AddHandler*-Befehl:

```
For i As Short = 1 To 26
  bnNeu = New Button()
  bnNeu.Tag = CType(i,String)
  AddHandler bnNeu.Click, AddressOf ButtonSammelClick()
  ButtonListe.Add (bnNeu)
  Me.Controls.Add (bnNeu)
Next
```

ButtonSammelClick ist eine Prozedur, die bei jedem Klick auf einen der Buttons aufgerufen wird. Über die *Tag*-Eigenschaft des Buttons, die über den *Sender*-Parameter übergeben wird, kann festgestellt werden, welcher Button angeklickt wurde:

```
ButtonNr = CType(Sender, Button).Tag
```

Damit steht die Funktionalität eines Steuerelementfeldes auch unter Visual Basic 2005 zur Verfügung, nur auf dem Formular zusammenstellen lässt es sich nicht[8].

Sie finden das Beispiel auf der Buch-CD in der Projektdatei 09_ButtonFeld.sln.

Das folgende Beispiel ordnet 26 Buttons in einem Panel an, sodass diese die Innenfläche des Panels optimal ausfüllen. In einer Schleife wird bei jedem Durchlauf ein neuer Button angelegt, mit Eigenschaftswerten ausgestattet und am Ende an die Controls*-Auflistung des Panels gehängt:*

```
Dim B As Button
Dim XPos, YPos As Single
YPos = 10
For i As Short = 1 To 26
  B = New Button
  With B
    .Text = Chr(64 + i)
    If XPos + .Width > Panel1.Width Then
```

8 Denkbar wäre es, einen Designer zu programmieren, der eine entsprechende Funktionalität ermöglicht. Der Aufwand würde aber in keinem Verhältnis zum Nutzen stehen.

```
      XPos = 10
      YPos += 32
    End If
    .Width = 36
    .Height = 24
    .Top = YPos
    .Left = XPos
    XPos += 48
  End With
  Panel1.Controls.Add(B)
  AddHandler B.Click, AddressOf ButtonClick
Next
```

Der gemeinsame Click*-Handler macht nichts anderes, als die* Text*-Eigenschaft des angeklickten Buttons auszugeben:*

```
Sub ButtonClick(ByVal Sender As Object, ByVal e As EventArgs)
  MessageBox.Show("Gib mir ein " & Sender.Text)
End Sub
```

Abbildung 9.24: Im Prinzip gibt es Steuerelementfelder auch bei Visual Basic 2005.

9.8 MDI-Anwendungen

Eine MDI-Anwendung ist eine reguläre Windows-Anwendung, bei der lediglich das Zusammenspiel der Fenster einfachen Regeln gehorcht, die als »Multiple Document Interface« (MDI) bezeichnet werden und schon »uralt« sind. Demnach

- besitzt eine Anwendung ein Hauptfenster, in dessen Innenfläche sich alle übrigen Fenster anordnen. Das Hauptfenster ist das MDI-Elternfenster, die übrigen Fenster stellen die MDI-Kindfenster dar (eine Anwendung kann aber auch mehrere MDI-Hauptfenster aufweisen).
- kann jedes Kindfenster nur in der Innenfläche des Hauptfensters verschoben werden. Wird ein Kindfenster auf Maximalgröße vergrößert, füllt es die Innenfläche des Elternfensters vollständig aus. Seine Titelleiste wird in der Titelleiste des Hauptfensters angezeigt.

- lassen sich alle Kindfenster überlappend, nebeneinander, untereinander und als Symbole verkleinert zusammenstellen.
- werden alle Kindfenster in einem *Fenster*-Menü aufgelistet.

MDI-Anwendungen sind immer dann sehr praktisch, wenn eine Anwendung mit einer beliebigen Anzahl an Dokumentfenstern arbeitet, die mit möglichst wenig Aufwand verwaltet werden sollen und die sich an bestimmte Regeln anpassen, die bei allen MDI-Anwendungen gleich sind. MDI-Anwendungen sind aber nur eine Option. Sie können das Thema komplett ignorieren, ohne etwas Wichtiges zu verpassen. MDI-Anwendungen besitzen aber nach wie vor ihre Berechtigung, sodass im Folgenden ein kleiner Bildbetrachter als MDI-Anwendung vorgestellt werden soll.

Sie finden das Beispiel auf der Buch-CD in der Projektdatei 09_MdiBildbetrachter.sln.

Ein Formular zum MDI-Hauptfenster machen

MDI-Formulare gibt es nicht mehr, ein Formular wird durch Setzen der *IsMdiContainer*-Eigenschaft auf *True* zum MDI-Hauptfenster.

Ein Formular zum MDI-Kindfenster machen

Eine Eigenschaft, die ein Formular zum MDI-Kindfenster macht, gibt es nicht. Ein Formular wird zum MDI-Kindfenster, indem seine *MdiParent*-Eigenschaft zur Laufzeit mit einem Verweis auf das MDI-Hauptfenster belegt wird.

Die folgende Click*-Prozedur eines ToolStrip-Buttons zeigt einen Dateiauswahldialog zum Laden einer Bitmapdatei an und stellt diese anschließend in der Innenfläche eines Formulars dar, das zunächst komplett neu angelegt und dann zum MDI-Kindfenster wird:*

```
Using Ofd As New OpenFileDialog
  With Ofd
    .InitialDirectory = Environment.GetFolderPath _
      (Environment.SpecialFolder.MyPictures)
    If .ShowDialog = Windows.Forms.DialogResult.OK Then
      Dim MdiForm As New Form
      With MdiForm
        .BackgroundImage = New Bitmap(Ofd.FileName)
        .BackgroundImageLayout = ImageLayout.Stretch
        .Text = Path.GetFileName(Ofd.FileName)
        .MdiParent = Me
        .Show()
      End With
    End If
  End With
End Using
```

Auf das aktive MDI-Kindfenster zugreifen

Das aktive MDI-Kindfenster steht über die Eigenschaft *ActiveMdiChild* des Formulars zur Verfügung, ein (*Form*-)Array mit allen MDI-Kindfenstern über die Eigenschaft *MdiChildren*.

Das folgende Beispiel invertiert das Hintergrundbild des aktiven MDI-Kindfensters (mithilfe einiger »GDI-Tricks«, genauer gesagt Klassen im Namespace System.Drawing.Imaging*, um die es in Kapitel 22 gehen wird.*


```
If Me.ActiveMdiChild IsNot Nothing Then
  Dim Imas As New ImageAttributes
  Dim Img As Image = Me.ActiveMdiChild.BackgroundImage
  Dim Bm As New Bitmap(Img.Width, Img.Height)
  Dim Cm As New ColorMatrix
  With Cm
    .Matrix00 = -1
    .Matrix11 = -1
    .Matrix22 = -1
  End With
  Imas.SetColorMatrix(Cm)
  Dim G As Graphics = Graphics.FromImage(Bm)
  G.DrawImage(Img, New Rectangle(0, 0, Img.Width, Img.Height), 0, _
    0, Img.Width, Img.Height, GraphicsUnit.Pixel, Imas)
  G.Dispose()
  Img.Dispose()
  Me.ActiveMdiChild.BackgroundImage = Bm
End If
```

Das Fenster-Menü implementieren

Das typische *Fenster*-Menü einer MDI-Anwendung, das die Namen aller MDI-Kindfenster enthält, lässt sich sehr einfach realisieren, wenngleich es beim neuen *MenuStrip*-Control gegenüber dem *MainMenu*-Control des .NET Frameworks 1.1 eine kleine Änderung gegeben hat: Damit ein Menü, also ein *MenuStrip*-Control, die Fensterliste anzeigt, muss die *MdiWindowListItem*-Eigenschaft des *MenuStrip*-Controls mit der Referenz auf jenes *ToolStripMenuItem*-Objekt gelegt werden, das die Fenster anzeigen soll. Heißt dieses zum Beispiel *mnFenster*, lautet der Befehl wie folgt:

```
MenuStrip1.MdiWindowListItem = mnFenster
```

Über die (mit .NET 2.0 eingeführte) *IsMdiWindowListEntry*-Eigenschaft lässt sich feststellen, ob ein *ToolStripMenuItem* Teil einer solchen Fensterliste ist oder nicht.

Das Anordnen der Kindfenster implementieren

Zum Schluss soll natürlich auch verraten werden, wie sich die MDI-Fenster »in einem Aufwasch« zum Beispiel nebeneinander oder überlappend anordnen lassen. Das erledigt die *LayoutMdi*-Methode des Formulars, welcher der Modus als Parameter übergeben wird. Der folgende Aufruf ordnet die Fenster nebeneinander an:

```
Me.LayoutMdi(MdiLayout.TileHorizontal)
```

Abbildung 9.25: Ein Bildbetrachter als typische MDI-Anwendung

9.9 Das fehlende Chart-Control

Abschließend muss leider auf eine kleine Lücke in dem ansonsten so umfangreichen Windows Forms-Modell hingewiesen werden. Aus welchen Gründen auch immer hat sich Microsoft dazu entschieden, kein Chart-Control mitzuliefern. Grundsätzlich ist das natürlich kein Problem, denn es gibt eine reichhaltige Auswahl an entsprechenden Controls. Das Dumme ist nur, dass die allermeisten nicht gerade preiswert sind (dafür aber auch einiges zu bieten haben). Doch es gibt eine Ausnahme: *Chart FX Lite* von Software FX. Laden Sie das Control über die URL *http://www.softwarefx.com/GetChartFXLite* herunter (Sie müssen sich dazu allerdings auf der Webseite registrieren und benötigen einen Schlüssel, der Ihnen per E-Mail zugesendet wird), und führen Sie die folgenden Schritte aus:

1. Legen Sie ein neues Projekt vom Typ WINDOWS-ANWENDUNG an.
2. Klicken Sie die Toolbox mit der rechten Maustaste an, und rufen Sie den Menüeintrag ELEMENT AUSWÄHLEN auf. Klicken Sie auf DURCHSUCHEN, und wählen Sie die Assembly *ChartFX.Lite.dll* im entsprechenden Verzeichnis aus (standardmäßig *C:\Programme\ChartFX Lite for .NET\bin*).

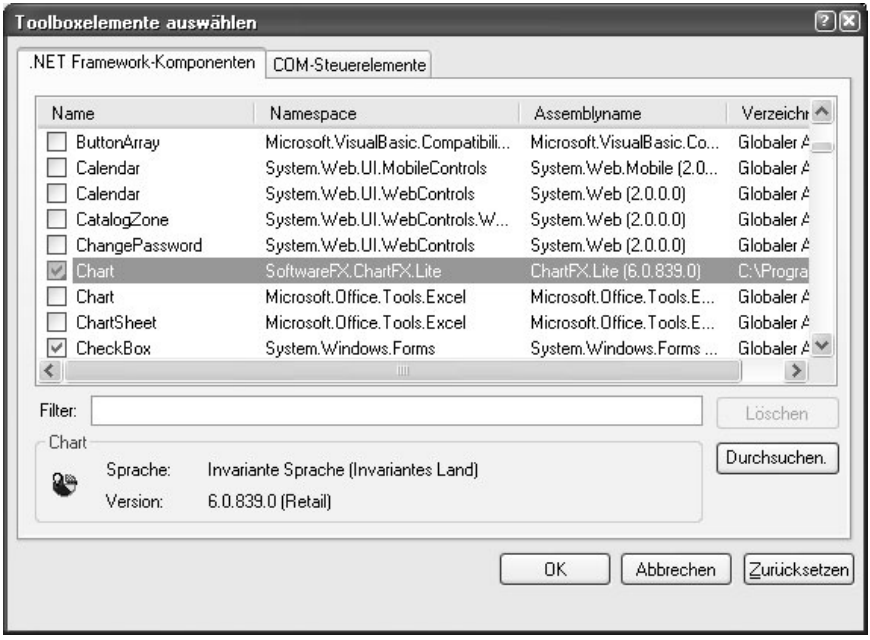


Abbildung 9.26: Das Chart-Control wird zur Toolbox hinzugefügt.

3. Bestätigen Sie die Auswahl mit OK, und ordnen Sie das Chart-Control auf dem Formular an. Dabei ist zu beachten, dass das Control eine gewisse Mindestgröße besitzen muss. Ziehen Sie das Control entsprechend groß auf, ansonsten wird später nur ein leerer Rahmen oder lediglich ein Teil der Legende angezeigt.
4. Fügen Sie in die *Form_Load*-Prozedur die folgenden Befehle ein, die ein paar Zufallszahlen erzeugen:

```
Chart1.Gallery = SoftwareFX.ChartFX.Lite.Gallery.Pie
Chart1.Chart3D = True
Chart1.LegendBox = True
For i As Short = 1 To 10
  Chart1.Value(0, i) = New Random().Next(1, 100)
Next
```

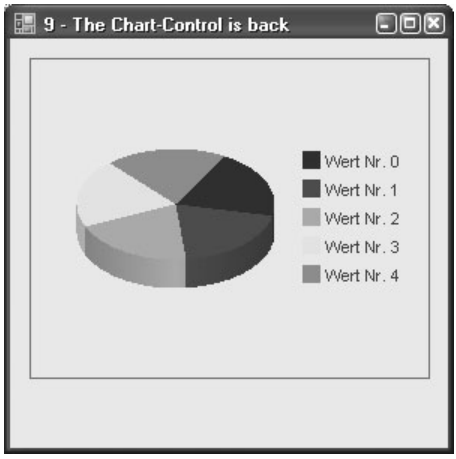


Abbildung 9.27: Das erfreut jeden Programmierer – ein kleines 3D-Kuchendiagramm.

9.10 Zusammenfassung

Die Toolbox von Visual Basic 2005 bietet (fast) alles, was benötigt wird, um moderne Benutzeroberflächen mit relativ wenig Aufwand zu gestalten. Im Unterschied zu VB6 stehen diese Möglichkeiten allen .NET-Sprachen gleichermaßen zur Verfügung. Zu den beiden Dingen, die fehlen, gehören ein Chart-Control und eine »Guide Line«, also eine Art »Offizielles Handbuch zur Gestaltung moderner Windows-Oberflächen«[9]. Der erste Punkt lässt sich relativ schnell abhaken, da es Dutzende von Chart-Controls anderer Anbieter gibt (unter anderem das kostenlose Chart FX Lite), der zweite Punkt ist in erster Linie eine Sache der Erfahrung und eines gewissen Gespürs für Gestaltung, Farben und Ergonomie.

9 Das schreibe ich, glaube ich, schon seit über zehn Jahren. Irgendjemand sollte sich das einmal zu Herzen nehmen.

PEARSON Education

Markt+Technik

ASP.NET 2.0

Leistungsfähige Webapplikationen
mit Visual Basic 2005

WENZ HAUSER SAMASCHKE KORDWIG TRENNHAUS

Markt+Technik

KOMPENDIUM

Einführung | Arbeitsbuch | Nachschlagewerk

Teil 2
Formulare und Steuerelemente

4 Formulare mit HTML Controls

Die wichtigsten Aufgaben bei jeder serverseitigen Webskriptsprache sind Abfrage, Auswertung und Verarbeitung von Formulardaten. Anwendungsgebiete hierfür gibt es viele:

- Feedback-Formulare, mit denen der Nutzer Rückmeldungen über die Website tätigen kann. Der Benutzer muss dazu nicht extra sein Email-Programm starten und unter Umständen nicht einmal seine Email-Adresse preisgeben. Damit wird eine wichtige Hemmschwelle überschritten und die Chancen, dass Sie wertvolle Rückmeldungen Ihrer Besucher erhalten, steigen.
- Support-Formulare, mit denen der Benutzer technische Anfragen stellen kann. Der Zwang, die Daten in verschiedene, exakt spezifizierte Formularelemente einzugeben, liefert Ihnen bei der Auswertung Vorteile; so kommen Sie schneller an die gewünschten Daten, als wenn Sie eine Freitext-Email interpretieren und dort die interessanten Inhalte extrahieren müssen.
- Web-Front-Ends (also Masken) für andere Anwendungen, beispielsweise Gästebücher.

ASP.NET bietet mehrere Möglichkeiten, auf Formulardaten zuzugreifen. In diesem und den folgenden Kapiteln werden wir alle vorstellen. Die erste Möglichkeit ist noch von ASP bekannt und wird hauptsächlich aus Gründen der Abwärtskompatibilität beibehalten. Die neuen Möglichkeiten von ASP.NET, insbesondere die mögliche strikte Trennung von Code und Content, bieten dem Programmierer weitere Ansätze der Formulargestaltung und -verarbeitung. Zum einen ist es möglich, bekannte HTML-Formularelemente serverseitig neu zu beleben, und zum anderen bietet ASP.NET neue, eigene Elemente, die in Formularcode umgesetzt werden.

Die Hauptanwendungen beim Formularzugriff sind folgende:

- Zugriff auf Formulardaten
- Vollständigkeitsüberprüfung
- Vorausfüllung, falls das Formular zuvor nicht komplett ausgefüllt wurde

Alle diese Punkte werden wir im Folgenden mit den verschiedenen Möglichkeiten von ASP.NET für Formulare behandeln. Zunächst stellen wir den »alten« Weg vor, auf Formulardaten mit ASP.NET zuzugreifen. Das funktioniert wunderbar, ist für einige Anwendungen immer noch sehr praktisch, nutzt aber einige der eingebauten Vorteile von ASP.NET nicht aus. In Abschnitt 4.2 erfahren Sie dann, wie ASP.NET eine Brücke zwischen der herkömmlichen Formularbehandlung und HTML zu schlagen versucht.

4.1 Formulardaten von Hand

Wer schon einmal mit ASP programmiert hat, weiß bereits, dass der Zugriff auf Formulardaten sehr einfach über ein spezielles Objekt von ASP erfolgt: das `Request`-Objekt. Generell kann über `Request("xyz")` (Visual Basic 2005) bzw. `Request["xyz"]` (C#) auf den Wert in dem Formularfeld zugegriffen werden, das als `name`-Attribut `"xyz"` hat.

Betrachten wir ein einfaches HTML-Textfeld:

```
<input type="text" name="Login" />
```

Der Wert in diesem Formularfeld steht nach dem Versand in `Request("xyz")` (VB 2005) bzw. `Request["xyz"]` (C#).

4.1.1 Versandmethode

Für den Versand von HTML-Formularen über das World Wide Web gibt es zwei gängige Methoden:

- GET und
- POST

Standardmäßig wird GET verwendet. Das bedeutet, dass die Formulardaten in der URL übergeben, also dort angehängt werden. Sie können das mit einer einfachen, statischen HTML-Seite ausprobieren. Nachfolgend finden Sie ein HTML-Formular mit ein paar Feldern:

Listing 4.1: Ein einfaches HTML-Formular ohne Skriptcode (formular.html)

```
<!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>
  <title>Formular</title>
</head>
<body>
  <form>
    Textfeld:
    <input type="text" name="Textfeld" />
    <br />
    Passwortfeld:
    <input type="password" name="Passwortfeld" />
    <br />
    Mehrzeiliges Textfeld
    <textarea name="Mehrzeilig"></textarea>
    <br />
    Checkbox
    <input type="checkbox" name="Checkbox" value="an" />
    <br />
    Radiobutton
    <input type="radio" name="Radio" value="r1" />1
    <input type="radio" name="Radio" value="r2" />2
    <br />
    Auswahlliste
    <select name="Auswahlliste" size="3" multiple="multiple">
```

```
      <option value="o1">Option 1</option>
      <option value="o2">Option 2</option>
      <option value="o3">Option 3</option>
    </select>
    <br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

In Abbildung 4.1 sehen Sie das Formular sowie einige Beispielwerte, die wir eingetragen haben. Wenn Sie das Formular verschicken, wird es neu geladen, aber die zuvor eingegebenen Formularwerte sind verschwunden.

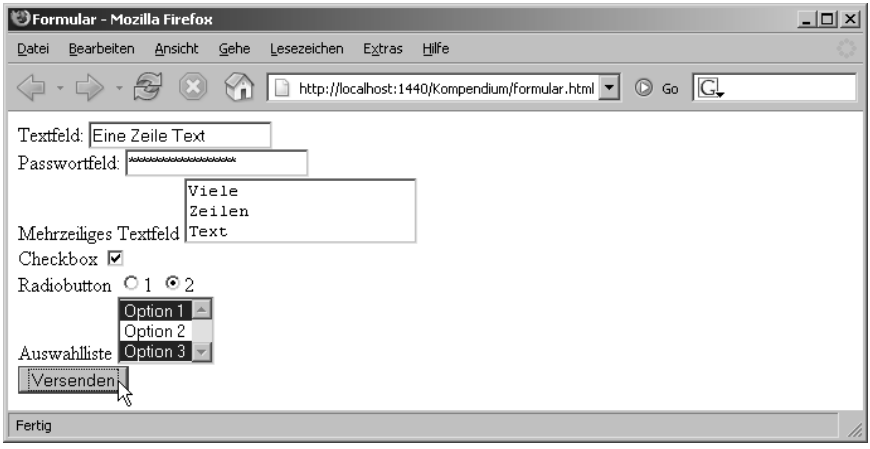


Abbildung 4.1: Das HTML-Formular mit ein paar Beispielwerten

Was ist passiert? Nun, die Formulardaten wurden per GET verschickt. Die Bezeichnungen der Formularfelder (das sind die `name`-Attribute) und die dort eingegebenen Werte wurden im Format `Name=Wert` an die URL angehängt. Die einzelnen dieser so genannten *Name-Wert-Paare* werden durch das kaufmännische Und (`&`) voneinander getrennt.

Hier die komplette URL, die beim Versand des obigen Formulars aufgerufen wurde:

```
http://localhost:1440/Kompendium/formular.html?Textfeld=Eine+Zeile+Text&Pass
wortfeld=v%F6llig+unsichtbar&Mehrzeilig=Viele%0D%0AZeilen%0D%0AText&Checkbox
=an&Radio=r2&Auswahlliste=o1&Auswahlliste=o3
```

Da dies ein wenig unübersichtlich ist, haben wir die URL in mehrere einzelne Zeilen aufgebrochen. Dazu schreiben wir jeweils daneben, welchem Formularwert welcher Ausschnitt der URL entspricht:

- `http://localhost:1440/Kompendium/formular.html` – der URL des Formulars
- `?Textfeld=Eine+Zeile+Text` – das einzeilige Textfeld
- `&Passwortfeld=v%F6llig+unsichtbar` – das Passwortfeld
- `&Mehrzeilig=Viele%0D%0AZeilen%0D%0AText` – das mehrzeilige Textfeld
- `&Checkbox=an` – die Checkbox
- `&Radio=r2` – der Radiobutton
- `&Auswahlliste=o1&Auswahlliste=o3` – die Auswahlliste

Wir können also festhalten:

- Name und Werte werden durch Gleichheitszeichen voneinander getrennt.
- Die einzelnen Name-Wert-Paare werden durch das kaufmännische Und (`&`) voneinander getrennt.
- Die ganzen Name-Wert-Paare werden mit einem vorangestellten Fragezeichen (?) an den Namen der Datei (hier: *formular.html*) angehängt.
- Sonderzeichen werden besonders maskiert. Aus Leerzeichen werden Pluszeichen, andere Sonderzeichen werden durch `%` und ihren hexadezimalen Zeichencode ersetzt. Beispielsweise hat ein Zeilensprung den Zeichencode 13, hexadezimal 0D. Also entspricht `%0D` einem Zeilensprung.

Dies ist der Versand per GET. Diese Methode ist zum Testen sehr bequem, sind doch aus der URL alle relevanten Daten ersichtlich. In der Praxis wird jedoch meistens auf GET verzichtet, von Suchmaschinen einmal abgesehen. GET hat nämlich eine Reihe von Nachteilen:

- Die Länge einer URL ist bei Browsern, Proxyservern und Webservern begrenzt. Einige Systeme erlauben nur maximal 500 Zeichen URL, aber die meisten Softwareprodukte machen spätestens bei 2000 Zeichen dicht.
- Aus der URL sind die kompletten Formulardaten ersichtlich. Diese URL wird in der History-Liste (Netscape) bzw. in der Verlaufsliste (Internet Explorer) des Browsers gespeichert und ist in Firmennetzwerken zumeist auch aus dem Proxy-Log ermittelbar. Sensible Daten wie Passwörter oder Kreditkartennummern sind somit unter Umständen von Dritten einsehbar.

POST hat diese Nachteile nicht. Hier werden die Formulardaten zunächst auch in Name-Wert-Paare umgewandelt. Allerdings werden diese Daten dann nicht an die URL angehängt, sondern in der HTTP-Anforderung an den Webserver hinter dem HTTP-Header untergebracht. Im obigen Beispiel könnte dann die HTTP-Anforderung beispielsweise folgendermaßen aussehen:

```
POST /Kompendium/formular.html HTTP/1.1
Host: localhost:1440
User-agent: Mozilla/47.11
Content-length: 286
Content-type: application/x-www-form-urlencoded

Textfeld=Eine+Zeile+Text&Passwortfeld=v%F6llig+unsichtbar&Mehrzeilig=Viele%0D%
   0AZeilen%0D%0AText&Checkbox=an&Radio=r2&Auswahlliste=o1&Auswahlliste=o3
```

Sie sehen also: Zunächst erscheinen die HTTP-Header-Informationen, dann eine Leerzeile und dann die Formulardaten.

Damit dies auch tatsächlich so funktioniert, müssen Sie im HTML-`<form>`-Tag den `method`-Parameter auf `"post"` setzen:

```
<form method="post">
```

Das `<form>`-Element kennt den Parameter `action`, in dem das Skript angegeben werden kann, an das die Formulardaten verschickt werden müssen. Wenn Sie jedoch den Parameter nicht angeben (wie in den vorherigen Beispielen geschehen), werden die Formulardaten an die aktuelle Datei verschickt. Diese »Abkürzung« werden wir in den nächsten Kapiteln noch häufiger verwenden.

Aber zurück zu ASP.NET und dem `Request`-Objekt. Im `Request`-Objekt selbst finden Sie sowohl POST- als auch GET-Daten und sogar Cookies (sowie weitere Werte, auf die wir an dieser Stelle nicht eingehen möchten). Im Sinne einer sauberen Entwicklung macht es jedoch sehr viel Sinn, explizit auf POST- oder auf GET-Werte zuzugreifen. Aus diesem Grund gibt es innerhalb des `Request`-Objekts Unterkollektionen, eine speziell für GET und eine speziell für POST:

- Über `Request.QueryString` greifen Sie auf GET-Daten zurück (die hinter dem Fragezeichen an eine URL angehängten Daten werden im Englischen als *Querystring* bezeichnet).
- Über `Request.Form` greifen Sie auf POST-Daten zurück. Das rührt daher, dass POST-Daten nur per Formular zustande kommen, während GET-Daten auch von Hand erzeugt worden sein könnten (indem einfach eine URL mit angehängtem Querystring eingegeben wird).

Die beiden Eselsbrücken, GET/Querystring und POST/Formular, sollten Ihnen helfen, Verwechslungen zu vermeiden.

Doch nun genug der langen Vorrede – werfen wir einen Blick auf die verschiedenen Formularfelder und wie Sie mit ASP.NET darauf zugreifen können.

4.1.2 Formularfelder

Im Allgemeinen können Sie über `Request.Form("xxx")` auf den Wert im Formularelement mit `name`-Attribut `"xxx"` zugreifen. Je nach Formularfeldtyp gibt es jedoch ein paar Besonderheiten; daher werden wir die einzelnen Feldtypen jeweils explizit aufführen und untersuchen.

Wir verwenden im Folgenden jeweils den Formularversand per POST. Wenn Sie stattdessen auf GET setzen, müssen Sie alle Vorkommen von `Request.Form` *durch* `Request.QueryString` *ersetzen.*

Textfeld

Ein Textfeld wird durch folgendes HTML-Element dargestellt:

```
<input type="text" name="Feldname" />
```

Über `Request.Form("Feldname")` können Sie dann auf den Text im Formularfeld zugreifen.

Im nachfolgenden Beispiel enthält das Listing ein Textfeld; nach dem Formularversand wird der eingegebene Text ausgegeben. Der zugehörige Code wird ausnahmsweise nicht in einem serverseitigen `<script>`-Block am Anfang der Seite ausgegeben, sondern direkt mitten auf der Seite, mit `<% ... %>`. Diese Art des »Spaghetti-Codes« ist mittlerweile verpönt und wird nur noch recht selten gebraucht (manchmal in Verbindung mit der Anzeige von Datenbankdaten), aber an dieser Stelle erläutert er recht schön das Konzept. Wie gesagt, die von Microsoft empfohlene Ansteuerung von Formulardaten folgt in Abschnitt 4.2.


CODE

Listing 4.2: Der Inhalt des Textfeldes wird ausgegeben (textfeld.aspx).

```
<%@ Page 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>Formular</title>
</head>
<body>
  <%  Response.Write(HttpUtility.HtmlEncode( _
    Request.Form("Feldname")))%>
  <form method="post">
    <input type="text" name="Feldname" /><br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```


HALT

Mit `HttpUtility.HtmlEncode()` *wandeln Sie gefährliche Sonderzeichen in der Eingabe in entsprechendes HTML um; so wird zum Beispiel aus der öffnenden spitzen Klammer* < *die zugehörige HTML-Entität* `<`*. Mehr Informationen zum Thema Sicherheit erhalten Sie in Kapitel 25.*


TIPP

Für einen schnelleren Zugriff auf die Methode `HtmlEncode` *können Sie den Namensraum* `System.Web.HttpUtility` *wie folgt importieren:*

```
<%@ Import Namespace="System.Web.HttpUtility" %>
```

Sie können dann direkt über `HtmlEncode` *auf die Methode zugreifen.*

Abbildung 4.2: Der eingegebene Wert wird über dem Textfeld angezeigt.

Bei den folgenden Beispielen können Sie analog testen; wir verzichten dort auf ausführlichere Erklärungen und setzen bei der Skriptsprache wieder verstärkt auf Visual Basic (2005).

Passwortfeld

Die HTML-Darstellung eines Passwortfeldes ist folgende:

```
<input type="password" name="Feldname" />
```

Auch hier können Sie über `Request.Form("Feldname")` (VB) bzw. über `Request.Form["Feldname"]` (C#) auf den Feldinhalt zugreifen. Hier ein komplettes Listing:

Listing 4.3: Der Inhalt des Passwortfeldes wird ausgegeben (passwortfeld.aspx).


```
<%@ Page 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>Formular</title>
</head>
<body>
  <%  Response.Write(HttpUtility.HtmlEncode( _
    Request.Form("Feldname")))%>
  <form method="post">
    <input type="password" name="Feldname" /><br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```


Mehrzeiliges Textfeld

Ein mehrzeiliges Textfeld wird in HTML durch `<textarea>` und `</textarea>` eingeschlossen:

```
<textarea name="Feldname"></textarea>
```

Der Text im mehrzeiligen Feld kann wie gehabt über `Request.Form("Feldname")` bzw. `Request.Form["Feldname"]` ermittelt werden, je nachdem, ob Sie VB oder C# einsetzen. Hier ein Beispiellisting in VB:

Listing 4.4: Der Inhalt des mehrzeiligen Feldes wird ausgegeben (mehrzeilig.aspx).


```
<%@ Page 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>Formular</title>
</head>
<body>
  <%Response.Write(HttpUtility.HtmlEncode( _
    Request.Form("Feldname")))%>
  <form method="post">
    <textarea name="Feldname"></textarea><br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

Checkbox

Eine Checkbox hat einen (eindeutigen) Namen und einen zugehörigen Wert. In HTML wird das durch die Parameter `name` und `value` ausgedrückt:

```
<input type="checkbox" name="Feldname" value="an" />
```

Nach dem Versand des Formulars enthält `Request.Form("Feldname")` (VB) bzw. `Request.Form["Feldname"]` (C#) den Wert `"an"`, wenn die Checkbox aktiviert worden ist[1], andernfalls enthält er eine leere Zeichenkette.

Listing 4.5: Der Name der Checkbox wird ausgegeben, falls aktiviert (checkbox.aspx).

```
<%@ Page 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>Formular</title>
</head>
<body>
  <%  Response.Write(HttpUtility.HtmlEncode( _
    Request.Form("Feldname")))%>
  <form method="post">
    <input type="checkbox" name="Feldname" value="an" /><br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

Radiobutton

Genau wie eine Checkbox ist auch ein Radiobutton entweder aktiviert (»angekreuzt«) oder nicht. Der große Unterschied ist, dass Radiobuttons in Gruppen unterteilt werden. Von allen Radiobuttons einer Gruppe kann immer nur maximal einer aktiviert sein. Es ist natürlich auch möglich, dass keiner der Radiobuttons aktiviert werden kann.

Wenn auf jeden Fall ein Radiobutton aktiviert werden soll, sollten Sie einen der Radiobuttons einer Gruppe vorauswählen. Die meisten Browser ermöglichen es dem Benutzer nicht, einen Radiobutton zu deselektieren. Die einzige Möglichkeit besteht darin, einen anderen Radiobutton aus derselben Gruppe zu aktivieren.

Hier der HTML-Code für einen Radiobutton:

```
<input type="radio" name="Feldname" value="Button" />
```

Ist dieser Radiobutton aktiviert, so enthält `Request.Form("Feldname")` den Wert `"Button"`. Wenn Sie C# verwenden, müssen Sie dementsprechend auf `Request.Form["Feldname"]` zurückgreifen.

1 Hätte die Checkbox kein `value`-Attribut besessen, wäre beim Ankreuzen als Wert `"on"` übertragen worden.

Der Name der Gruppe von Radiobuttons wird im `name`-Attribut angegeben. Alle Radiobuttons mit dem gleichen `name`-Attribut gehören zu einer Gruppe und nur einer dieser Buttons kann aktiviert werden. Die einzelnen Radiobuttons unterscheiden sich also anhand des `value`-Attributs.

Hier ein Listing, in dem der Name (in diesem Fall das `value`-Attribut) des ausgewählten Radiobuttons ausgegeben wird:

Listing 4.6: Der Name des gewählten Radiobuttons wird ausgegeben (radio.aspx).


```
<%@ Page 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>Formular</title>
</head>
<body>
  <%Response.Write(HttpUtility.HtmlEncode( _
    Request.Form("Feldname")))%>
  <form method="post">
    1<input type="radio" name="Feldname" value="Button 1" />
    2<input type="radio" name="Feldname" value="Button 2" />
    3<input type="radio" name="Feldname" value="Button 3" />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

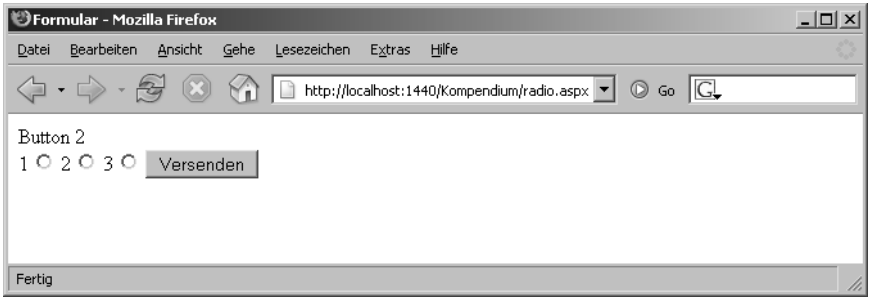



Abbildung 4.3: Der mittlere Radiobutton war beim Versand aktiviert.

Auswahlliste

In einer Auswahlliste, auch Drop-down-Menü genannt, kann der Benutzer einen oder mehrere Einträge auswählen, je nachdem, welche Einstellung im HTML-Code verwendet wird. Es gibt die folgenden Möglichkeiten:

- Der folgende Code erzeugt eine Drop-down-Liste, aus der ein Element ausgewählt werden kann:

```
<select name="Feldname">
  <option value="Element 1">1. Element</option>
  <option value="Element 2">2. Element</option>
  <option value="Element 3">3. Element</option>
</select>
```

- Der folgende Code erzeugt eine Auswahlliste mit drei Elementen (davon alle zu Anfang sichtbar); trotzdem kann nur eines oder keines der Elemente ausgewählt werden:

```
<select name="Feldname" size="3">
  <option value="Element 1">1. Element</option>
  <option value="Element 2">2. Element</option>
  <option value="Element 3">3. Element</option>
</select>
```

- Folgender Code erzeugt schließlich eine Auswahlliste der Höhe 3; mithilfe der Tasten ⇧ und Strg (sowie der Maus) können beliebig viele Elemente ausgewählt werden:

```
<select name="Feldname" size="3" multiple="multiple">
  <option value="Element 1">1. Element</option>
  <option value="Element 2">2. Element</option>
  <option value="Element 3">3. Element</option>
</select>
```

Der Zugriff auf das oder die ausgewählten Elemente geschieht wie bei den vorherigen Formularfeldern auch über `Request.Form("Feldname")` bzw. `Request.Form["Feldname"]`. Es ist nun interessant zu untersuchen, welcher Unterschied bei Einfach- und Mehrfachlisten besteht; letztere Listen sind jene mit dem Attribut `multiple` im `<select>`-Tag. Das nachfolgende Listing bietet daher zwei Auswahllisten an.

Listing 4.7: Die Namen der gewählten Listenelemente werden ausgegeben (liste.aspx).

```
<%@ Page 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>Formular</title>
</head>
<body>
  <%
    Response.Write("Liste 1: " & _
      HttpUtility.HtmlEncode(Request.Form("Liste 1")))
    Response.Write("<br />")
    Response.Write("Liste 2: " & _
      HttpUtility.HtmlEncode(Request.Form("Liste 2")))
  %>
  <form method="post">
    <select name="Liste 1">
      <option value="Element 1">1. Element</option>
      <option value="Element 2">2. Element</option>
      <option value="Element 3">3. Element</option>
    </select>
    <br />
    <select name="Liste 2" size="3" multiple="multiple">
      <option value="Element 1">1. Element</option>
      <option value="Element 2">2. Element</option>
      <option value="Element 3">3. Element</option>
    </select>
    <br />
```

```
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

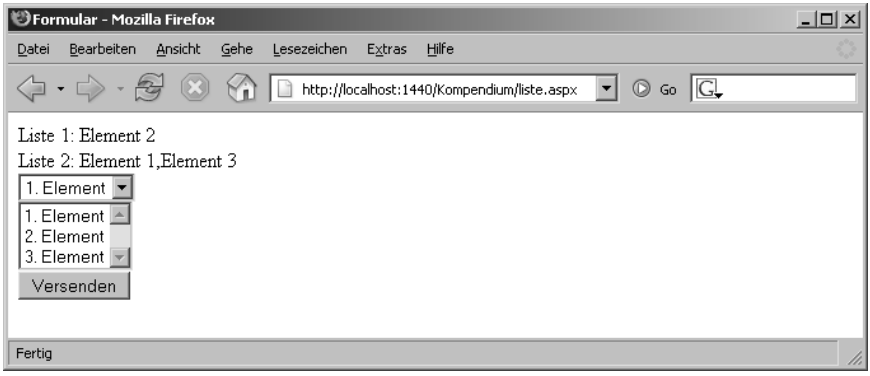


Abbildung 4.4: Die ausgewählten Elemente werden angezeigt.

Wie Sie in Ihrem Webbrowser oder in Abbildung 4.4 sehen können, wird jeweils der oder die `value`-Parameter der entsprechenden Listenelemente ausgegeben. Bei mehreren Elementen (also bei `<select multiple="multiple">`) werden diese Werte durch Kommata voneinander getrennt.

Wenn Sie bei Ihren Listenelementen (`<option>`-Element) das `value`-Attribut weglassen, übermitteln die meisten Browser an seiner Stelle die Beschriftung des Elements, also den Text zwischen `<option>` und `</option>`. Verlassen Sie sich aber nicht darauf – und setzen Sie immer den `value`-Parameter.

Datei-Upload

Ein oft vergessenes Formularfeld ist das zum Upload von Dateien:

```
<input type="file" name="Feldname" />
```

Mit den herkömmlichen Mitteln von ASP war es nicht möglich, auf diese Formularwerte zuzugreifen. Es kursierten zwar einige mögliche Lösungen im Web, die aber zumindest bei größeren Dateianhängen allesamt versagten. Für diesen Zweck musste eine Third-Party-Komponente angeschafft werden. ASP.NET bietet hierfür einen Ausweg. Dazu benötigen Sie aber Techniken, die erst in Abschnitt 4.2 vorgestellt werden; über das `Request`-Objekt direkt ist das nicht bzw. nur mit großem Aufwand möglich.

Damit haben Sie einen Überblick über alle relevanten Formularfeldtypen erhalten.

Die folgenden Feldtypen haben unter anderem gefehlt:

- `<input type="hidden" />` – *versteckte Formularfeld, funktioniert analog zu Textfeldern.*
- `<input type="submit" />` – *Versendeschaltfläche, wird in der Regel nicht serverseitig abgefragt, funktioniert aber analog zu Textfeldern.*

- `<input type="image" />` – *Versendegrafik, funktioniert wie eine Versendeschaltfläche, übergibt aber gleichzeitig in* `<Name>.x` *und* `<Name>.y` *die relativen Koordinaten des Mausklicks.*
- `<input type="button" />`, `<input type="reset" />` – *Diese Schaltflächen lösen keinen Formularversand aus.*

4.1.3 Ausgabe aller Formularangaben

Als letztes Beispiel wollen wir alle Daten im Formular ausgeben. Diese recht trivial klingende Aufgabe ist in ähnlicher Form Bestandteil vieler Skripte. Anstelle der Ausgabe der einzelnen Werte werden Sie im Praxisbetrieb die angegebenen Daten beispielsweise in einer Datenbank abspeichern.

Die naheliegendste Möglichkeit besteht darin, für jedes Formularfeld von Hand `Request.Form("Feldname")` bzw. `Request.Form["Feldname"]` auszugeben. Wir wollen an dieser Stelle einen bequemeren, aber nicht ganz so flexiblen Weg gehen. Per `For-Each`-Schleife bzw. `for-in`-Schleife werden alle Formularwerte ausgegeben.

Hier das Codestück, wie es mit Visual Basic realisiert werden könnte:

```
Dim element As String
For Each element In Request.Form
  Response.Write("<b>" & _
    HttpUtility.HtmlEncode(element) & _
    ":</b> ")
  Response.Write( _
    HttpUtility.HtmlEncode(Request.Form(element)))
  Response.Write("<br />")
Next
```

Der komplette VB-Code sieht dann folgendermaßen aus – wir verwenden ein spartanisches Formular mit allen wichtigen Feldtypen:

Listing 4.8: Alle Formulardaten werden ausgegeben (ausgabe.aspx)

```
<%@ Page 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>Formular</title>
</head>
<body>
  <%
    Dim element As String
    For Each element In Request.Form
      Response.Write("<b>" & _
        HttpUtility.HtmlEncode(element) & _
        ":</b> ")
      Response.Write( _
        HttpUtility.HtmlEncode(Request.Form(element)))
      Response.Write("<br />")
    Next
  %>
```

```
  <form method="post">
    Textfeld:
    <input type="text" name="Textfeld" />
    <br />
    Passwortfeld:
    <input type="password" name="Passwortfeld" />
    <br />
    Mehrzeiliges Textfeld
    <textarea name="Mehrzeilig"></textarea>
    <br />
    Checkbox
    <input type="checkbox" name="Checkbox" value="an" />
    <br />
    Radiobutton
    <input type="radio" name="Radio" value="r1" />1
    <input type="radio" name="Radio" value="r2" />2
    <br />
    Auswahlliste
    <select name="Auswahlliste" size="3" multiple="multiple">
      <option value="o1">Option 1</option>
      <option value="o2">Option 2</option>
      <option value="o3">Option 3</option>
    </select>
    <br />
    <input type="submit" value="Versenden" />
  </form>
</body>
</html>
```

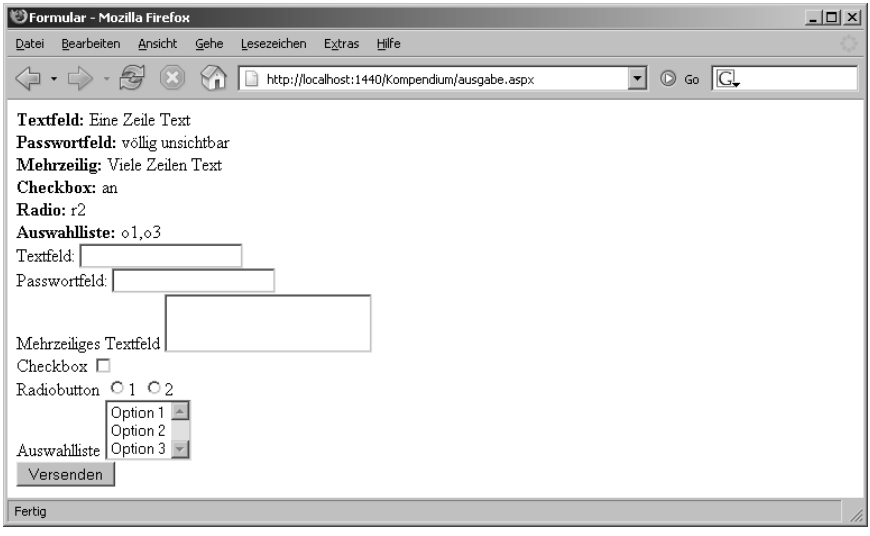



Abbildung 4.5: Die Formulardaten werden per Schleife ausgegeben.

Nun ist es aber in der Regel so, dass Sie nur die Formulardaten ausgeben möchten, das Formular jedoch nicht. Ein Ansatz besteht darin, als Ziel des Formulars (Parameter `action`) eine eigene *.aspx*-Seite zu verwenden. Dies hat jedoch den Nachteil, dass Sie beim Fehlen von Pflichtfeldern das Formular nicht erneut anzeigen können – Sie befinden sich ja mittlerweile auf einer anderen ASP.NET-Seite.

Aus diesem Grund wird zumeist ein anderes Vorgehen gewählt. Zunächst erhält die Schaltfläche zum Verschicken des Formulars ein `name`-Attribut, was normalerweise nicht erforderlich ist:

```
<input type="submit" name="Submit" value="Versenden" />
```

Der Vorteil: Wenn das Formular verschickt wird, enthält `Request.Form("Submit")` (bzw. bei C# `Request.Form["Submit"]`) den Wert `"Versenden"`. So kann also einfach überprüft werden, ob das Formular gerade verschickt wurde (dann: Formulardaten ausgeben) oder nicht (dann: das nackte Formular anzeigen):

```
If Request.Form("Submit") = "Versenden" Then
  ' Formulardaten per For-Each-Schleife ausgeben
Else
  ' Formular ausgeben
End If
```

Nachfolgend das entsprechende VB-Listing:

Listing 4.9: Entweder das Formular oder die Daten werden ausgegeben (ausgabe2.aspx).

```
<%@ Page 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>Formular</title>
</head>
<body>
  <%
    If Request.Form("Submit") = "Versenden" Then
      Dim element As String
      For Each element In Request.Form
        Response.Write("<b>" & element & ":</b> ")
        Response.Write( _
          System.Web.HttpUtility.HtmlEncode( _
            Request.Form(element)))
        Response.Write("<br />")
      Next
    Else
  %>
  <form method="post">
    Textfeld:
    <input type="text" name="Textfeld" />
    <br />
    Passwortfeld:
    <input type="password" name="Passwortfeld" />
    <br />
    Mehrzeiliges Textfeld
    <textarea name="Mehrzeilig"></textarea>
    <br />
    Checkbox
    <input type="checkbox" name="Checkbox" value="an" />
    <br />
    Radiobutton
    <input type="radio" name="Radio" value="r1" />1
```

```
    <input type="radio" name="Radio" value="r2" />2
    <br />
    Auswahlliste
    <select name="Auswahlliste" size="3" multiple="multiple">
      <option value="o1">Option 1</option>
      <option value="o2">Option 2</option>
      <option value="o3">Option 3</option>
    </select>
    <br />
    <input type="submit" name="Submit" value="Versenden" />
  </form>
  <%
  End If
  %>
</body>
</html>
```

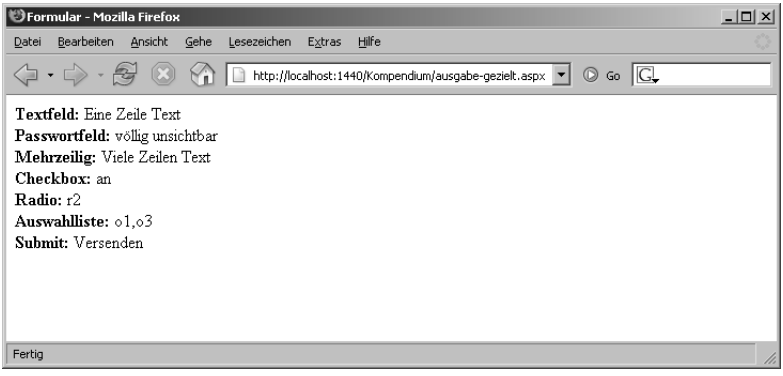


Abbildung 4.6: Diesmal werden nach dem Versand nur die Formulardaten angezeigt.

Wo sind die Umbrüche aus dem mehrzeiligen Feld hin? Die sind zwar immer noch da, doch ein Umbruch in HTML wird im Browser als Leerzeichen angezeigt. Sie müssen also alle Zeilensprünge (in .NET durch `System.Environment.NewLine` *dargestellt) durch das entsprechende HTML-Markup (*`<br />`*) ersetzen.*

Die nächste nahe liegende Anwendung besteht darin, die Formulardaten zu prüfen: Sind alle Felder ausgefüllt? Sind die eingegebenen Werte sinnvoll? Doch hier selbst Hand anzulegen und das zu programmieren wäre unsinnig, denn das ASP.NET Framework hat solche Standard-Szenarien abgebildet. Mehr zum Thema Validierung erfahren Sie in Kapitel 6, die dazu notwendigen Grundlagen über die Funktionsweise von Steuerelementen in ASP.NET erhalten Sie im nächsten Abschnitt und in Kapitel 5.

Wenn wir ein kleines Zwischenfazit ziehen möchten: Die Abfrage von Formulardaten mit ASP.NET geht relativ simpel, genau wie in den meisten anderen Webtechnologien von Haus aus auch, sei es PHP oder Cold Fusion oder was auch immer. Der Vorteil des alles umspannenden Frameworks ist hier aber zunächst vergeben. Nur auf HTTP-Daten zuzugreifen ist lediglich ein kleiner Teilaspekt einer modernen Webanwendung. Die Formularfelder selbst sollten (wie oben abgesprochen) validiert werden, wenn das fehlschlägt, ist eine Vorausfüllung fällig. Viele weitere Anforderungen ergeben sich in alltäglichen Praxisprojekten. Der nächste Abschnitt stellt deswegen den Ansatz von ASP.NET vor, das Arbeiten mit HTML-Formularen etwas zu vereinfachen und auf eine solide und objektorientierte Basis zu stellen.

4.2 Grundlegendes zu HTML Controls

In diesem Buch werden Sie immer wieder sehen, dass Sie im Kopf der *.aspx*-Seite innerhalb der Funktion `Page_Load()` auf HTML-Elemente weiter unten zugreifen können, etwa nach folgendem Muster:

```
Sub Page_Load
  ausgabe.InnerText = "ASP.NET macht Spaß"
End Sub
```

Dabei ist `ausgabe` gleichzeitig der Wert des `id`-Parameters eines (fast beliebigen) HTML-Elements weiter unten in der Seite. Durch `runat="server"` wird der ASP.NET-Interpreter angewiesen, dieses HTML-Element serverseitig zu verarbeiten:

```
<p id="ausgabe" runat="server"></p>
```

Nach Ausführung des obigen Codes wird an den Browser ein `<p>`-Element geschickt, das den angegebenen Text (`"ASP.NET macht Spaß"`) enthält. Diese Form der speziellen HTML-Elemente wird **HTML Controls** genannt und funktioniert im Wesentlichen für jedes HTML-Element. Einzige Voraussetzung: Sie benötigen eine ID und ein `runat="server"`.

Es liegt natürlich nahe, auch für Formularelemente HTML Controls zu verwenden; alleine die Aufgabe der Vorausfüllung der Felder scheint damit einfacher zu sein als noch im vorherigen Abschnitt.

Die HTML Controls für Formularelemente (und auch andere Elemente, wenngleich diese nur wenig Funktionalität bieten) sind alle im Namespace `System.Web.UI.HtmlControls` untergebracht. Wenn Sie die Dokumentation aus dem .NET Framework SDK verwenden, können Sie eine schnelle Übersicht über die einzelnen Klassen in diesem Namespace erhalten (siehe Abbildung 4.7).

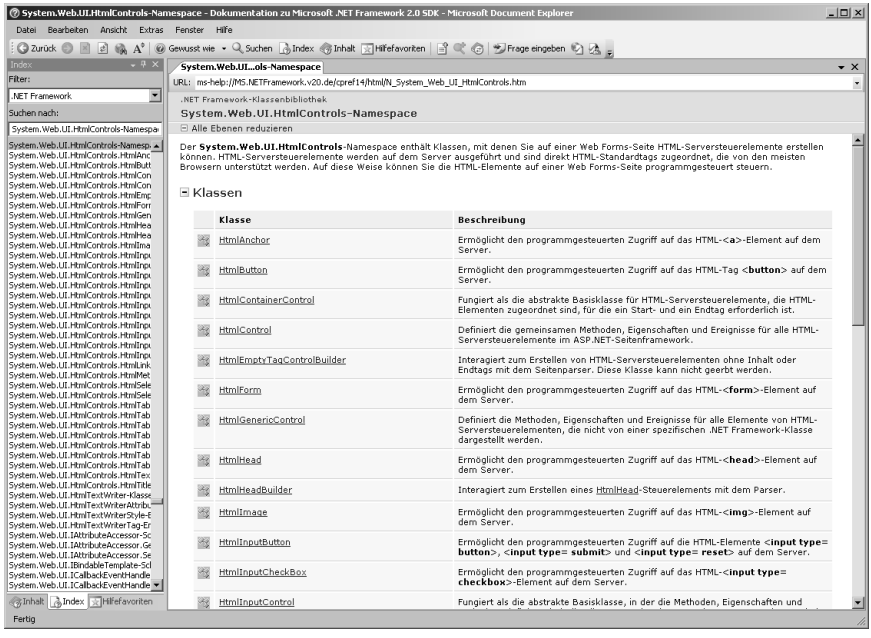


Abbildung 4.7: Die Klassen in `System.Web.UI.HtmlControls`

Die Basisklasse für alle HTML-Elemente (die ein `runat="server"` aufweisen) ist `HtmlGenericControl`. Ein Blick in den Klassen-Browser zeigt hier die Ereignisse und Eigenschaften auf, die jedes HTML-Element unter ASP.NET unterstützt. Die wichtigsten sind hierbei `InnerText` und `InnerHtml`. Damit können Sie den Inhalt eines HTML Controls sowohl auslesen als auch setzen.

Wie der Name schon andeutet, wird durch `InnerHtml` der HTML-Code innerhalb des Elements repräsentiert; `InnerText` steht für den Text (ohne HTML-Formatierungen), der in dem Element steht.

Betrachten wir folgendes Beispiel:

```
<p id="absatz" runat="server">
<b>&lt;%@ Page Language="C#" %&gt;</b>
</p>
```

Die Eigenschaft `absatz.InnerHtml` ist offensichtlich der HTML-Code innerhalb des `<p>`-Elements, also Folgendes:

```
<b>&lt;%@ Page Language="C#" %&gt;</b>
```


Welchen Wert hat aber nun `InnerText`? Nun, HTML-Tags, die nicht als »Klartext« im Browser angezeigt werden, werden natürlich ignoriert. Die spitzen Klammern, im HTML-Code noch HTML-codiert (`<` und `>`), werden durch das jeweilige Ausgabezeichen ersetzt, hier also < und >. Damit hat `InnerText` folgenden Wert:

```
<%@ Page Language="C#" %>
```

Besonders interessant wird der Unterschied jedoch erst beim Setzen der Eigenschaften. Wenn Sie eine lange Zeichenkette haben und sie nicht gesondert mit `HtmlEncode` vorbehandeln möchten, setzen Sie `InnerText` und der ASP.NET-Prozess erledigt automatisch die Konvertierung für Sie. Wollen Sie stattdessen HTML-Formatierungen verwenden, wie beispielsweise in obigem Codeausschnitt Fettdruck (`<b>...</b>`), dann müssen Sie `InnerHtml` setzen und die entsprechenden HTML-Auszeichnungen verwenden.

Da es in diesem Kapitel aber um Formulare geht, werfen wir nun einen genaueren Blick auf die Formular-HTML-Controls. Eines der allgemeineren Controls ist `HtmlInputControl`, das die Oberklasse für alle mit dem `<input>`-Element erzeugten Formularfelder ist (also einzeilige Textfelder, Passwortfelder, Radiobuttons, Checkboxen, versteckte Formularfelder, File-Uploads, Versende-Grafiken und Versende-Schaltflächen). Die Dokumentation offenbart hier alle zur Verfügung stehenden Eigenschaften, wobei insbesondere die Eigenschaft `Value` interessant ist; sie enthält den Wert im entsprechenden Formularfeld.

Später in diesem Kapitel werden wir ausführlich auf die einzelnen relevanten Formularfeldtypen und ihre Umsetzung in ASP.NET-HTML-Controls eingehen.

Die zur Verfügung stehenden Klassen innerhalb von `System.Web.UI.HtmlControls` können Sie Tabelle 4.1 entnehmen[2].

2 Zwei Klassen, die nur programmativ, aber nicht deklarativ, also mit ASP.NET-Markup verwendet werden können, wurden herausgelassen.

Tabelle 4.1: Die Klassen für HTML Controls

Klasse	Entsprechendes HTML-Element	Beschreibung
HtmlAnchor	<a>	Link
HtmlButton	<button>	Schaltfläche
HtmlContainerControl		HTML-Container (kann weitere HTML-Elemente enthalten)
HtmlControl	alle hier vorgestellten Elemente	Allgemeine Oberklasse
HtmlForm	<form>	Formular
HtmlGenericControl	diverse	Klasse für HTML-Elemente ohne eigene Klasse (z.B.)
HtmlHead	<head>	Kopfbereich der Seite
HtmlImage	<img>	Grafik
HtmlInputButton	<input type="button"> <input type="submit">	Formular-Schaltfläche
HtmlInputCheckbox	<input type="checkbox">	Checkbox
HtmlInputControl	<input>	Oberklasse für <input>
HtmlInputFile	<input type="file">	File-Upload
HtmlInputHidden	<input type="hidden">	Verstecktes Formularfeld
HtmlInputImage	<input type="image">	Versende-Grafik
HtmlInputRadioButton	<input type="radio">	Radiobutton
HtmlInputText	<input type="text"> <input type="password">	Einzeiliges Eingabefeld (auch Passwortfeld)
HtmlLink	<a>	Link
HtmlMeta	<meta>	Meta-Tag
HtmlSelect	<select>	Auswahlliste
HtmlTable	<table>	Tabelle
HtmlTableCell	<td>	Tabellenzelle
HtmlTableCellCollection	<td> (mehrfach)	Mehrere Tabellenzellen
HtmlTableRow	<tr>	Tabellenzeile
HtmlTableRowCollection	<tr> (mehrfach)	Mehrere Tabellenzeilen
HtmlTextArea	<textarea>	Mehrzeiliges Textfeld
HtmlTitle	<title>	Seitentitel

Die Namen der einzelnen Controls sind für Sie insbesondere dann interessant, wenn Sie eine Eigenschaft nachschlagen möchten, beispielsweise in der Online-Referenz oder – natürlich noch besser und unserer Meinung nach übersichtlicher – im Referenzteil dieses Buchs. Bei der Programmierung selbst werden Ihnen diese Klassennamen nicht begegnen, Sie greifen auf Eigenschaften und Ereignisse zu, vermutlich immer auf dieselben.

4.3 Formularversand mit HTML Controls

Bevor wir nun direkt einsteigen noch ein wichtiger Hinweis: Alle Formular-HTML-Controls müssen innerhalb eines serverseitigen Formulars stehen, also innerhalb von `<form runat="server">...</form>`. Zusätzliche Parameter, beispielsweise den Namen des Formulars oder die Versandmethode, müssen Sie nicht angeben; das macht ASP.NET im Alleingang.

4.3.1 Formular serverseitig

Betrachten Sie folgende minimalistische ASP.NET-Seite:

```
<form runat="server" />
```

Wenn Sie eine Datei mit dieser einen Zeile erstellen, ihr die Endung `.aspx` geben und sie im Webbrowser aufrufen, erhalten Sie deutlich längeren Code zurück (etwas optisch aufgehübschte Wiedergabe):

```
<form name="ctl00" method="post" action="skript.aspx" id="ctl00">
  <div>
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="/wEPDwUJMjk3MDAwMjUwZGS2glfCpWO7aVv/z8uDaVipJ6HehA==" />
  </div>
</form>
```

Zunächst hat das Formular also einen Namen (`name`-Parameter) bekommen, `"_ctl00"`. Dann wurde die Versandmethode auf POST gesetzt (`method="post"`). Schließlich wurde noch das Ziel des Formulars gesetzt, und zwar auf das aktuelle Skript (`action="skript.aspx"`). Daran sehen Sie, dass wir das aus einer Zeile bestehende Testskript *skript.aspx* genannt haben.

Sie werden ebenfalls überrascht feststellen, dass das Formular ein verstecktes Feld namens `__VIEWSTATE` enthält (brav, von ASP.NET XHTML-konform in einem `<div>`-Element platziert); als Wert ist eine kryptische, in diesem Beispiel 48 Zeichen lange Zeichenkette angegeben. ASP.NET benötigt diese Zeichenkette, um auf dieser Basis auf eingegebene Formulardaten ohne Cookies zugreifen und Formularfelder vorausfüllen zu können. Anhand der Zeichenkette weiß der ASP.NET-Interpreter, wo die Formulareingaben auf dem Server temporär abgelegt worden sind. Die Auswirkung dieses `__VIEWSTATE`-Feldes sehen Sie an späterer Stelle in diesem Kapitel noch.

Was aber passiert nun, wenn Sie Eigenschaften wie Formularnamen, Versandmethode und Versandziel selbst setzen möchten? Auf ein Neues, diesmal wird folgender Einzeiler (aus drucktechnischen Gründen auf zwei Zeilen aufgeteilt) getestet:

```
<form name="Formular" method="get"
      action="skript2.aspx" runat="server" />
```

Das Ergebnis sehen Sie hier:

```
<form name="form1" method="get" action="skript.aspx" id="form1">
  <div>
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
      value="/wEPDwUJMjk3MDAwMjUwZGS2glfCpWO7aVv/z8uDaVipJ6HehA==" />
  </div>
</form>
```

Sie sehen also – ASP.NET besitzt einen eigenen Willen. Lediglich die Versandmethode GET wurde beibehalten, der Rest wurde eliminiert. Jetzt ist Ihnen vermutlich schon klar, wieso wir im vorherigen Kapitel den herkömmlichen Zugriff auf Formulare so ausführlich erklärt haben. Zwar nimmt Ihnen ASP.NET eine Menge Arbeit ab, Sie verlieren damit aber auch einen Teil Ihrer Flexibilität als Programmierer.

*Es gibt noch eine weitere Einschränkung, die gerne vergessen oder verdrängt wird: Sie können pro ASP.NET nur **ein** serverseitiges Formular verwenden. Mit einem »serverseitigen Formular« meinen wir ein Formular mit* `runat="server"`.

4.3.2 Versand ermitteln

Wie zuvor gesehen, ist das Ziel des Formularversands immer die aktuelle Seite. Es spielt sich also alles, Formularausgabe und -verarbeitung, auf einer *.aspx*-Seite ab. So ähnlich haben wir das übrigens auch im vorherigen Kapitel und vermutlich auch Sie in Ihren bisherigen ASP-Projekten gehandhabt.

Es ist also notwendig festzustellen, ob die ASP.NET-Seite »frisch« aufgerufen wird oder gerade Formulardaten verschickt werden. Wir wollen an dieser Stelle drei Ansätze untersuchen, um festzustellen, ob ein Formular verschickt worden ist. Falls ja, wird eine entsprechende Meldung ausgegeben, andernfalls wird das Formular angezeigt. Nicht jeder der Ansätze führt übrigens zum Erfolg.

Das Formular selbst besteht nur aus einer Versende-Schaltfläche, um das Ganze einfach und übersichtlich zu halten:

```
<form runat="server">
  <input type="submit" value="Versenden"
         runat="server" />
</form>
```

Eine Warnung, die wir gar nicht oft genug anbringen können: Vergessen Sie auf keinen Fall das `runat="server"` *bei allen Formularelementen und natürlich auch bei dem Formular selbst. HTML Input Controls für Formulare werden nur innerhalb von serverseitigen Formularen unterstützt. Wenn Sie ein solches Element außerhalb eines serverseitigen Formulars einsetzen oder umgekehrt (also ein »normales« Formularelement innerhalb eines serverseitigen Formulars), erhalten Sie keine Fehlermeldung. Bei unerklärlichen Fehlern sollten Sie an dieser Stelle zuerst suchen!*

Schlag ins Wasser: Überprüfung der Versende-Schaltfläche

Der naheliegendste, aber nicht wirklich funktionierende Weg (dazu später mehr) besteht wie im vorherigen Kapitel auch darin, der Schaltfläche einen Namen zu geben und diesen dann abzufragen. Der erste Unterschied zu *ASP Classic* (ASP 1.0-3.0) ist zunächst der, dass Sie hierfür nicht mehr das `name`-Attribut verwenden dürfen, sondern das `id`-Attribut nehmen. Damit folgt Microsoft einer Empfehlung des W3C, den Standardisierungshütern des Internets. Die Schaltfläche sieht also folgendermaßen aus:

```
<input id="Submit" type="submit" value="Versenden"
       runat="server" />
```

In der Funktion `Page_Load`, die beim Laden der ASP.NET-Seite immer ausgeführt wird, können Sie dann über die ID (`"Submit"`) den Wert (`Value`) abfragen:

```
If Submit.Value = "Versenden" Then
  ' Formular wurde verschickt
Else
  ' Formular wurde nicht verschickt (also ausgeben)
End If
```

Hier ein Listing, das dies in die Tat umsetzen will. Wir definieren zusätzlich noch ein `<p>`-Element für die Ausgabe des Ergebnisses nach dem Versand:

```
<p id="ausgabe" runat="server" />
```

Sie müssen alle serverseitigen Tags (also Tags mit `runat="server"`) wieder schließen, sonst beschwert sich die ASP.NET-Engine. Wenn wir beispielsweise den Schrägstrich am Ende des obigen `<p>`-Elements nicht setzen, würde die Fehlermeldung aus Abbildung 4.8 angezeigt werden.

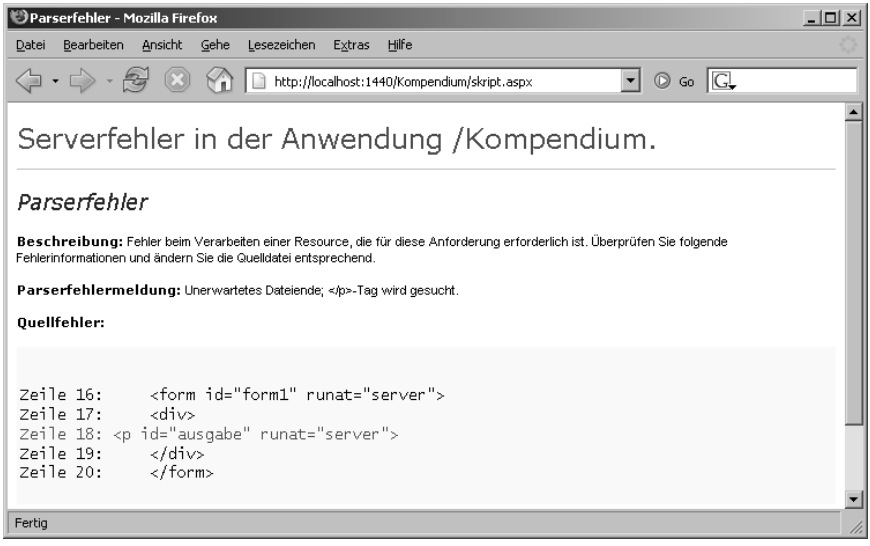


Abbildung 4.8: Fehlermeldung bei fehlendem Tag-Ende

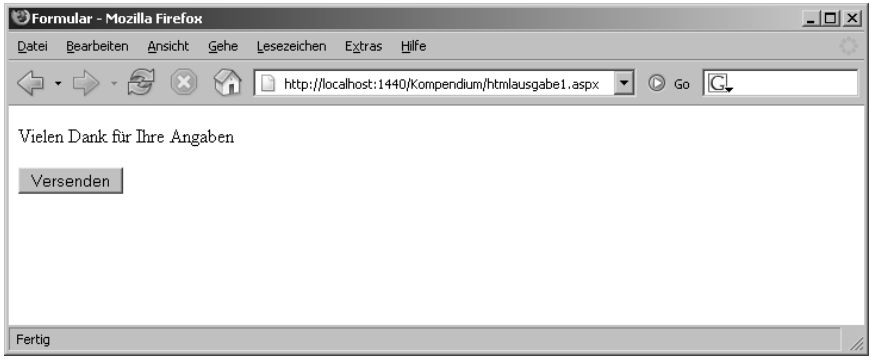
Hier das komplette Listing:

Listing 4.10: Der erste Ansatz – aber er funktioniert nicht (htmlausgabe1.aspx).

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Page_Load()
    If Submit.Value = "Versenden" Then
      ausgabe.InnerText = "Vielen Dank für Ihre Angaben"
    End If
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input id="Submit" type="submit" value="Versenden" runat="server" />
  </form>
</body>
</html>
```

Im Browser sehen Sie jedoch schon beim ersten Aufrufen der Seite eine Dankesmeldung (siehe Abbildung 4.9).

Abbildung 4.9: Die Ausgabe beim ersten Aufruf



Eine Erklärung ist schnell gefunden. Sie greifen auf `Submit.Value` zu, also den Wert im Formularelement mit `id="Submit"`. Die Funktion `Page_Load` wird allerdings erst beim Laden der Seite ausgeführt, bevor sie an den Browser geschickt wird. An dieser Stelle des Ablaufs sind alle Formularelemente schon erstellt worden, es gibt bereits die Versende-Schaltfläche mit `id="Submit"`. Dementsprechend hat `Submit.Value` schon beim ersten Laden den Wert `"Versenden"`. Diese Lösung kann also nicht funktionieren und wir müssen nach Alternativen suchen.

Schon besser: Die Eigenschaft IsPostBack

Die `Page`-Klasse beschreibt die aktuelle ASP.NET-Seite. Wenn Sie beispielsweise im Kopf einer Seite die Angabe der verwendeten Skriptsprache näher betrachten, finden Sie diese Klasse dort wieder:

```
<%@ Page Language="VB" %>
```

Die Klasse `Page` *befindet sich in der Assembly* `System.Web.UI`.

Die `Page`-Klasse hat die Eigenschaft `IsPostBack`. Diese ist genau dann `True` (bzw. `true`, bei C#), wenn gerade ein serverseitiges Formular auf die aktuelle Seite verschickt wurde. Diesen Vorgang nennt man *PostBack*. Damit ist die Überprüfung, ob ein Formular gerade verschickt worden ist, sehr einfach:

```
If Page.IsPostBack Then
  ' Formular wurde verschickt
Else
  ' Formular wurde nicht verschickt (also ausgeben)
End If
```


Hier ein komplettes Beispiel:

Listing 4.11: Überprüfung mit `Page.IsPostBack` (htmlausgabe2.aspx)


```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Page_Load()
    If Page.IsPostBack Then
      ausgabe.InnerText = "Vielen Dank für Ihre Angaben"
    End If
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input id="Submit" type="submit" value="Versenden" runat="server" />
  </form>
</body>
</html>
```

Alternativ: Serverseitige Funktion

Wieder einmal werfen wir einen Blick in einen Klassenbrowser (oder in den Anhang B), und zwar insbesondere auf die Klasse `System.Web.UI.HtmlControls.HtmlInputButton`. Diese ist wie zuvor erläutert für Schaltflächen zuständig, insbesondere also auch für Versende-Schaltflächen. Im Class Browser ist das Event (Ereignis) `ServerClick` zu sehen. Es wird aktiviert, wenn die Schaltfläche angeklickt wird.

Events funktionieren ähnlich wie ihre JavaScript-Pendants. Vor den Namen des Events wird ein "`On`" gesetzt. Damit erhalten Sie den Namen des Parameters, über den Sie den so genannten *Event-Handler*, also die Behandlungsfunktion beim Eintreten des Ereignisses angeben können.

In JavaScript kann das beispielsweise folgendermaßen aussehen:

```
<html>
<body onload="alert('Das ist clientseitig ...');">
</body>
</html>
```

Das Ereignis heißt hier `load` und tritt beim Laden der HTML-Seite im Browser ein. Sie können also durch `onload` angeben, was passieren soll, wenn das `load`-Ereignis eintritt. In diesem Fall wird ein modales Dialogfenster ausgegeben (siehe Abbildung 4.10).

Abbildung 4.10: Ein (clientseitig) erzeugtes Infofenster

Serverseitig funktioniert das ähnlich; folgendermaßen können Sie beim Eintreten des Ereignisses `ServerClick` für die Schaltfläche eine Behandlungsroutine angeben:

```
<input type="submit" value="Versenden"
       OnServerClick="Versand" runat="server" />
```

Wenn der Benutzer auf die Schaltfläche klickt, wird die serverseitige Funktion `Versand()` aufgerufen. Beachten Sie, dass Sie dies als Programmierer einstellen, sich aber nicht mit der tatsächlichen HTML-Ausgabe beschäftigen müssen. Im Browser selbst wird eine normale Schaltfläche dargestellt und das Formular per Mausklick an den Webserver geschickt. Der Endnutzer bekommt die Funktion `Versand()` gar nicht erst zu Gesicht, er weiß nicht einmal etwas von ihrer Existenz!

Bei der Erstellung der Funktion `Versand()`müssen Sie beachten, dass diese Funktion zwei Parameter enthält. Als erster Parameter wird eine Referenz auf das Objekt übergeben, das den Funktionsaufruf angestoßen hat; in diesem Fall also die Schaltfläche. Der zweite Parameter sind zusätzliche Argumente, die an das Ereignis übermittelt wurden (beispielsweise bei anderen Elementen die Koordinaten des auslösenden Mausklicks).

In der Regel benötigen Sie diese zwei Parameter nicht. Sie müssen sie zwar im Funktionskopf angeben, aber Sie werden sie innerhalb der Funktion nicht verwenden. Wir werden später noch eine Einsatzmöglichkeit aufzeigen.

Im Übrigen heißt der Parameter genau deswegen `onserverclick` *und nicht* `OnClick`*, weil* `onclick` *ein HTML-Event-Handler ist und für die Ausführung von clientseitigem Skriptcode zuständig ist. Durch* `onserverclick` *wird eine Namenskollision vermieden.*

Der Funktionskopf sieht unter VB folgendermaßen aus:

```
Sub Versand(o As Object, e As EventArgs)
  ' ...
End Sub
```

Nachfolgend nun das komplette Beispiel:

Listing 4.12: Überprüfung mit einer serverseitigen Funktion (htmlausgabe3.aspx)


```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Versand(ByVal o As Object, ByVal e As EventArgs)
    ausgabe.InnerText = "Vielen Dank für Ihre Angaben"
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Die Ergänzung von `ByVal` *bei den beiden Parametern für die Prozedur* `Versand()` *wird übrigens von Visual Web Developer bei der Eingabe automatisch hinzugefügt, ist aber für die eigentliche Funktionalität nicht notwendig.*

Fazit

Von den drei vorgestellten Methoden funktionieren nur zwei. In der Regel bevorzugen wir die letzte Methode. Der Hauptgrund liegt darin, dass der Code zur Formularverarbeitung in eine gesonderte Funktion ausgelagert werden kann, was den Code übersichtlicher macht. Der zweite Grund hat nichts mit Performance oder Effektivität zu tun, sondern hängt mit den technischen Gegebenheiten beim Buchdruck zusammen. Wenn wir `Page_Load` verwenden, benötigen wir eine `If`-Abfrage (nämlich die von `Page.IsPostBack`), die den folgenden Code um zwei zusätzliche Leerzeichen einrückt. Damit rückt jedoch auch gleichzeitig der rechte Seitenrand näher, wir versuchen allerdings Umbrüche möglichst zu vermeiden. Aber für Sie wird vermutlich nur der Hauptgrund zutreffen. Ansonsten sind beide Möglichkeiten als gleichwertig anzusehen.

4.3.3 Das Formular ausblenden

Wenn die Formulardaten verschickt (und verarbeitet) worden sind, wollen Sie womöglich das Formular nicht mehr anzeigen. Auch hier gibt es wieder mehrere Möglichkeiten.

Zunächst einmal könnten Sie nach dem Formularversand den Benutzer auf eine andere Seite umleiten, die eine Dankesmeldung oder Ähnliches enthält:

```
Sub Versand(o As Object, e As EventArgs)
  Response.Redirect("danke.aspx")
End Sub
```

Durch den Aufruf von `Response.Redirect ()` wird der Browser des Benutzers auf die als Parameter übergebene URL weitergeleitet, in diesem Fall `danke.aspx`.

Es gibt jedoch eine bequemere Möglichkeit, bei der Sie die aktuelle Seite nicht verlassen müssen. Zuerst müssen Sie dem Formular selbst auch eine ID vergeben (falls nicht eh schon automatisch von Visual Web Developer/Visual Studio gemacht), um es ansprechen zu können:

```
<form id="form1" runat="server">
...
</form>
```

Die zugehörige Klasse in `System.Web.UI.HtmlControls` ist `HtmlForm`. Dort gibt es die boolesche Eigenschaft `Visible`, die angibt, ob das Formular sichtbar ist oder nicht. Wenn Sie `Visible` auf `False` (bzw. bei C#: `false`, wobei auch VB die Kleinschreibung akzeptieren würde) setzen, wird das Formular nicht angezeigt. Hier ein komplettes Listing:

Listing 4.13: Nach dem Versand wird das Formular unsichtbar gemacht (htmlausgabe4.aspx).

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Versand(ByVal o As Object, ByVal e As EventArgs)
    ausgabe.InnerText = "Vielen Dank für Ihre Angaben"
    form1.Visible = False
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

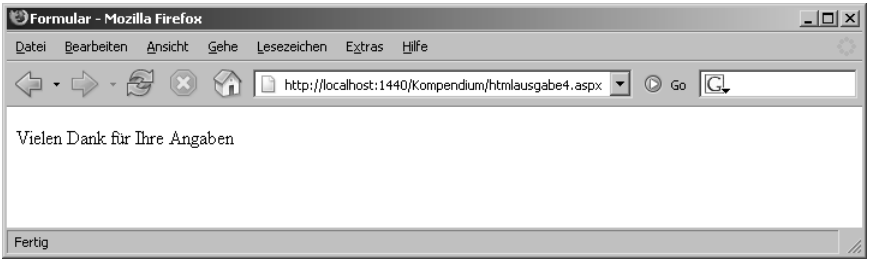


Abbildung 4.11: Das Formular wird nicht (mehr) angezeigt.

Wenn Sie den HTML-Code nach dem Versand betrachten, sieht er ungefähr folgendermaßen aus:

```
<!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><title>
  Formular
</title></head>
<body>
  <p id="ausgabe">Vielen Dank f&#252;r Ihre Angaben</p>
</body>
</html>
```

Das Formular ist also nicht nur unsichtbar, es ist verschwunden. Der ASP.NET-Interpreter schickt per `Visible = False` unsichtbar gemachte Elemente erst gar nicht an den Browser.

Sie können das Formular auch unsichtbar machen, indem Sie es in einen `<div>`*-Container legen:*

```
<div id="Container" runat="server">...</div>
```

Die `visibility`*-Stileigenschaft des Formulars setzen Sie dann serverseitig auf* `"hidden"`*:*

```
Container.Style("visibility") = "hidden"
```

Das Formular wird nun zwar an den Browser geschickt; wenn dieser aber Stylesheets unterstützt, sieht es der Benutzer nicht.

4.4 HTML Controls im Einsatz

Wie schon im vorhergehenden Kapitel, werden wir auch an dieser Stelle die Formularfeldtypen einzeln untersuchen und dabei jeweils aufzeigen, wie Sie auf die dort angegebenen Daten zugreifen können. Die Vorgehensweise ist immer ähnlich, der Teufel steckt aber sprichwörtlich im Detail, was nach einer genauen Darstellung verlangt.

4.4.1 Textfeld

Ein einzeiliges Textfeld können Sie mit dem `<input>`-Element darstellen; aber vergessen Sie hier (und bei den anderen Formularfeldtypen) nicht, `runat="server"` anzugeben!

```
<input type="text" id="Feldname" runat="server" />
```

Der Text in dem Textfeld steht in der Eigenschaft `Value`. Sie können sich das recht einfach merken, wenn Sie daran denken, dass Sie mit dem HTML-Parameter `value` ein Textfeld mit einem Wert vorbelegen können.

Folgendes Beispiellisting stellt ein Textfeld dar und gibt nach dem Versand den eingegebenen Text aus:

Listing 4.14: Der Wert aus dem Textfeld wird ausgegeben (htmltextfeld.aspx).

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Versand(ByVal o As Object, ByVal e As EventArgs)
    ausgabe.InnerText = "Ihre Eingabe: " & Feldname.Value
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input type="text" id="Feldname" runat="server" />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

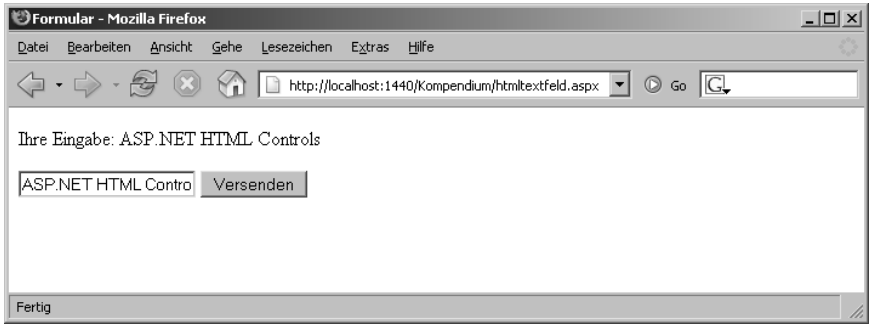


Abbildung 4.12: Der eingegebene Wert wird wieder ausgegeben.

Abbildung 4.12 können Sie entnehmen, dass der ASP.NET-Interpreter die Eingabe im Textfeld beibehält. Das Geheimnis liegt hier in dem versteckten Formularfeld, das Sie auch in der Ausgabe des Skripts *htmltextfeld.aspx* wieder finden:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwULLTExODM1OTUxNzUPZBYCAgQPFgIeCWlubmVyaHRtbAUjSWhyZSBFaW5nYWJlO
iBBU1AuTkVUIEhUTUwgQ29udHJvbHNkZH/mnFISQV9N8x4yttI/4PgSldbi" />
```

Wie bereits erläutert, werden über dieses Feld Formulareingaben beibehalten. Es war also kein zusätzlicher Code mehr nötig, um das Formular vorauszufüllen.

Sie können natürlich das Feld bei Bedarf mit einem anderen Wert füllen, indem Sie die Eigenschaft `Value` *setzen. In Visual Basic 2005 kann das dann so aussehen:*

```
Feldname.Value = "Vorausfüllung"
```

Diese Vorausfüllung lässt sich nicht deaktivieren, da die Daten bei jedem POST-Versand erneut an das Skript übergeben werden und der ASP.NET-Interpreter sie jedes Mal wieder in die entsprechenden Formularfelder einfügt. Wenn Sie das Formularfeld leeren möchten, müssen Sie es auf eine leere Zeichenkette setzen:

```
Feldname.Value = ""
```

Zwar schlagen einige Dokumentationen noch vor, den Parameter `EnableViewState` des entsprechenden Formularelements auf `"False"` zu setzen, das aber genügt nicht.

4.4.2 Passwortfeld

Passwortfelder werden – wie die meisten Formularfeldtypen – mit dem HTML-Element `<input>` dargestellt:

```
<input type="password" id="Feldname" runat="server" />
```

Auch hier steht in der Eigenschaft `Value` des Elements der angegebene Wert. Hier ein illustratives Beispiel:

Listing 4.15: Der Wert aus dem Passwortfeld wird ausgegeben (htmlpasswortfeld.aspx).


```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Versand(ByVal o As Object, ByVal e As EventArgs)
    ausgabe.InnerText = "Ihre Eingabe: " & Feldname.Value
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
```

```
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input type="password" id="Feldname" runat="server" />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

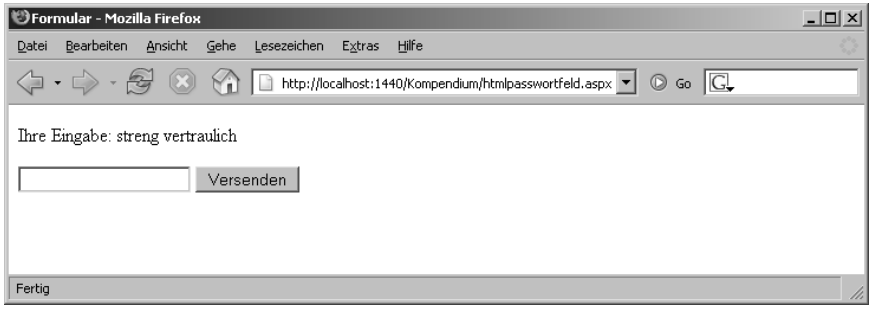


Abbildung 4.13: Der Wert aus dem Passwortfeld wird ausgegeben.

Sie sehen in Abbildung 4.13, dass das Passwortfeld nicht vorausgefüllt ist. Offensichtlich können Passwortfelder aus Sicherheitsgründen nicht vorbelegt werden. Wir haben das im vorherigen Abschnitt schon einmal kurz analysiert – das Passwort wäre dann im Klartext aus dem Cache des Webbrowsers einsehbar, was ein potenzielles Sicherheitsrisiko darstellen würde (obwohl normalerweise nur das eigene Windows-Konto Zugriff auf den Browsercache hat). Schlimmer noch: Das Passwort würde dann eventuell erneut ungesichert im Internet übertragen, und das ist ein tatsächliches Problem. Also: Keine Vorausfüllung bei Passwortfeldern.

4.4.3 Mehrzeiliges Textfeld

Das mehrzeilige Textfeld wird über `<textarea>` realisiert, also nicht mit dem `<input>`-Element.

```
<textarea id="Feldname" runat="server"></textarea>
```

Auch wenn das `<textarea>`-Element keinen HTML-Parameter `value` kennt (der Inhalt des Formularfelds steht hier zwischen `<textarea>` und `</textarea>`), können Sie vonseiten ASP.NET über die Eigenschaft `Value` auf den Wert im mehrzeiligen Textfeld zugreifen. Folgendes Listing illustriert dieses Vorgehen:

Listing 4.16: Der Feldwert wird ausgegeben (htmlmehrzeilig.aspx).

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Versand(ByVal o As Object, ByVal e As EventArgs)
    ausgabe.InnerText = "Ihre Eingabe: " & Feldname.Value
  End Sub
</script>
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <textarea id="Feldname" runat="server"></textarea>
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

4.4.4 Checkbox

Wie die meisten anderen Formularfelder auch, wird eine Checkbox mit dem `<input>`-Element dargestellt – der `type`-Parameter macht den Unterschied:

```
<input type="checkbox" id="Feldname" value="an"
       runat="server" />
```

Zwar gibt es auch bei Checkboxen von ASP.NET-Seite her eine Eigenschaft `Value`, sie nimmt aber immer den Wert des `value`-Parameters der Checkbox an, egal ob sie aktiviert ist oder nicht. Was Sie jedoch interessiert, ist der Zustand der Checkbox. Hier ist die Eigenschaft `Checked` geeignet. Bei `True` ist sie aktiviert, bei `False` dagegen nicht.

Listing 4.17: Es wird ausgegeben, ob die Checkbox angekreuzt (aktiviert) wurde (htmlcheckbox.aspx).


```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Versand(ByVal o As Object, ByVal e As EventArgs)
    ausgabe.InnerText = "Angekreuzt: " & Feldname.Checked
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <input type="checkbox" id="Feldname" value="an"
           runat="server" />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

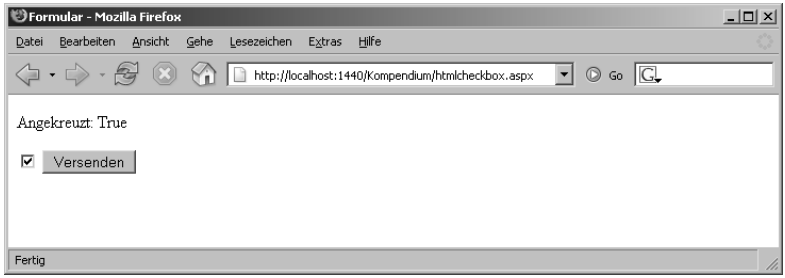


Abbildung 4.14: Der Zustand der Checkbox wird ausgegeben.

4.4.5 Radiobutton

Radiobuttons treten ja immer in Rudeln auf, wobei alle zusammengehörigen Radiobuttons denselben Namen (`name`-Parameter) tragen (aber unterschiedliche Werte, d.h. `value`-Parameter haben). Bei HTML Controls benötigen Sie jedoch den `id`-Parameter. Das Vorgehen ist nun folgendes:

- Setzen Sie den `name`-Parameter wie gewohnt, d.h. alle Radiobuttons aus einer Gruppe haben denselben Wert.
- Anstelle des `value`-Parameters verwenden Sie jedoch den `id`-Parameter.

Das sieht dann beispielsweise folgendermaßen aus:

```
1<input type="radio" name="Feldname" id="Button1"
        runat="server" />
2<input type="radio" name="Feldname" id="Button2"
        runat="server" />
3<input type="radio" name="Feldname" id="Button3"
        runat="server" />
```

Achten Sie darauf, im `id`*-Parameter keine Sonderzeichen und auch keine Leerzeichen zu verwenden. Unter HTML und ASP war das kein Problem, bei ASP.NET ist es jedoch eines.*

Folgender Code (vorausgesetzt natürlich, er befindet sich innerhalb von `<form runat="server">...</form>`) wird vom ASP.NET-Interpreter in folgenden Code umgesetzt:

```
1<input value="Button1" name="Feldname" id="Button1" type="radio" />
2<input value="Button2" name="Feldname" id="Button2" type="radio" />
3<input value="Button3" name="Feldname" id="Button3" type="radio" />
```

Sie sehen also – der `value`-Parameter wird automatisch eingesetzt, Sie brauchen sich darum nicht zu kümmern.

Um nun den Zustand der einzelnen Radiobuttons abzufragen, können Sie auf die Eigenschaft `Checked` zugreifen. Wie auch bei Checkboxen können Sie damit erkennen, ob ein Radiobutton aktiviert ist oder nicht.

Wenn Sie jedoch schnell feststellen möchten, welcher Radiobutton aus einer Gruppe aktiviert wurde, können Sie auf `Request.Form("name-Attribut")` *zugreifen.*

Nachfolgendes Listing verwendet sowohl `Request.Form` als auch die Eigenschaft `Checked` der HTML Controls:

Listing 4.18: Die einzelnen Radiobuttons werden untersucht (htmlradio.aspx).


```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Versand(ByVal o As Object, ByVal e As EventArgs)
    ausgabe.InnerHtml = "<ul><li>Button1: " & _
                        Button1.Checked & "</li>" & _
                        "<li>Button2: " & _
                        Button2.Checked & "</li>" & _
                        "<li>Button3: " & _
                        Button3.Checked & "</li>" & _
                        "</ul>Aktiviert: " & _
                        Request.Form("Feldname")
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    1<input type="radio" name="Feldname" id="Button1" runat="server" />
    2<input type="radio" name="Feldname" id="Button2" runat="server" />
    3<input type="radio" name="Feldname" id="Button3" runat="server" />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

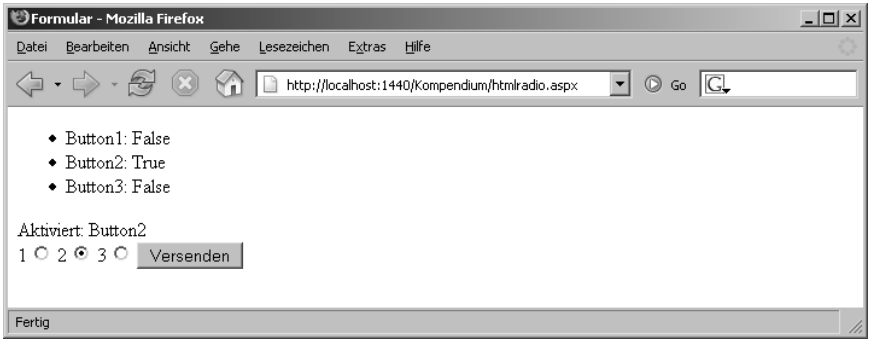


Abbildung 4.15: Die Zustände der einzelnen Radiobuttons

4.4.6 Auswahlliste

Das letzte Formularelement ist wieder das komplizierteste: die Auswahllisten. Das Problem liegt hier wie zuvor nicht bei den einfachen Auswahllisten, sondern bei den mehrfachen Auswahllisten (also denen mit `multiple`-Attribut im `<select>`-Tag). Aber der Reihe nach.

Beginnen wir mit den einfachen Auswahllisten:

```
<select name="Auswahlliste" size="3" runat="server">
  <option value="o1">Option 1</option>
  <option value="o2">Option 2</option>
  <option value="o3">Option 3</option>
</select>
```

Ein Blick in die Referenz beschert für Auswahllisten (Klasse `HtmlSelect`) eine ganze Reihe von Eigenschaften. Für den Zugriff auf das gewählte Element gibt es zwei Möglichkeiten:

- Sie greifen auf die Eigenschaft `Value` zu, die den Wert des `value`-Parameters des gewählten Listenelements enthält:
  ```
  Feldname.Value
  ```
- Oder Sie verwenden die Eigenschaft `SelectedIndex`, die den numerischen Index des gewählten Listenelements ausgibt (Achtung: Zählung beginnt bei 0). Diesen Index verwenden Sie, um über die Kollektion `Items` auf das entsprechende Element zuzugreifen. Bei diesem Element erhalten Sie dann über `Value` seinen Wert:
  ```
  Feldname.Items(Feldname.SelectedIndex).Value
  ```

Die letztere Methode ist natürlich viel umständlicher. Außerdem gibt es Probleme, wenn gar kein Element ausgewählt wurde. Dann hat `SelectedIndex` den Wert `-1` und der Zugriff auf `Feldname.Items(-1)`schlägt natürlich fehl. Sie müssen also eine zusätzliche Abfrage einführen.

Ein anderer Knackpunkt ist die Mehrfach-Auswahlliste:

```
<select id="Feldname" multiple="multiple" size="3" runat="server">
  <option value="o1">Option 1</option>
  <option value="o2">Option 2</option>
  <option value="o3">Option 3</option>
</select>
```

Zunächst ein großes Ärgernis an ASP.NET 2.0 an sich. Wenn Sie die Liste wie oben gezeigt im Browser aufrufen möchten, erhalten Sie die in Abbildung 4.16 gezeigte Fehlermeldung.

Der Grund: ASP.NET setzt die Auswahlliste intern in ein Objekt um, das unter anderem die Eigenschaft `Multiple` besitzt. Diese Eigenschaft ist ein boolescher Wert, also erwartet ASP.NET eine entsprechende Angabe. Das Folgende funktioniert – auch wenn es Visual Web Developer als fehlerhaft unterringelt:

```
<select id="Feldname" multiple="true" size="3" runat="server">
  <option value="o1">Option 1</option>
  <option value="o2">Option 2</option>
  <option value="o3">Option 3</option>
</select>
```

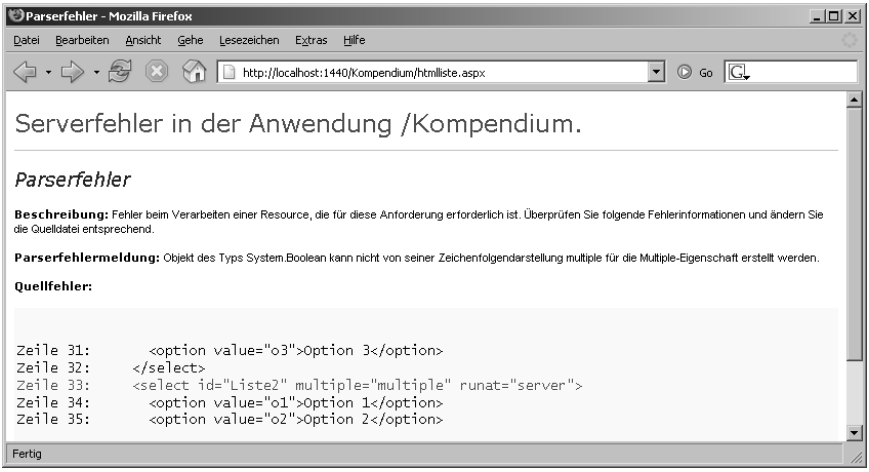


Abbildung 4.16: ASP.NET hat ein großes Problem ...

Doch zurück zur Programmierung an sich: Bei Mehrfachlisten enthält die Eigenschaft `SelectedIndex` lediglich die Position des ersten gewählten Listenelements; auf alle weiteren können Sie so nicht zugreifen. Hier können Sie sich behelfen, indem Sie per Schleife alle Elemente durchlaufen und dann überprüfen, ob das jeweilige Element ausgewählt wurde oder nicht.

Dazu brauchen Sie noch die folgenden Informationen:

- Die Anzahl der Listenelemente erhalten Sie über die Eigenschaft `Count` der Elemente der Auswahlliste (in unserem Beispiel: `Feldname.Items.Count`).
- Ob ein Element ausgewählt wurde oder nicht, sehen Sie anhand der booleschen Eigenschaft `Selected`:

  ```
  Feldname.Items(0).Selected
  ```

Die folgende Schleife durchläuft also die gesamte Liste und gibt aus, welche Elemente ausgewählt wurden:

```
For i As Integer = 0 To Feldname.Items.Count - 1
  If Feldname.Items(i).Selected Then
    Response.Write(Feldname.Items(i).Value & "<br />")
  End If
Next
```

Alternativ können Sie natürlich auch auf eine `For-Each`*-Schleife setzen.*

Hier ein Listing, das sowohl Einfach- als auch Mehrfach-Auswahllisten enthält:

Listing 4.19: Zwei verschiedene Auswahllisten (htmlselect.aspx)

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<script runat="server">
  Sub Versand(ByVal o As Object, ByVal e As EventArgs)
    ausgabe.InnerHtml = "Liste1: " & Liste1.Value
    ausgabe.InnerHtml += " (Element " & _
                         Liste1.SelectedIndex & ")<br />"
    ausgabe.InnerHtml += "Liste2: "
    For i As Integer = 0 To Liste2.Items.Count - 1
      If Liste2.Items(i).Selected Then
        ausgabe.InnerHtml += _
          HttpUtility.HtmlEncode(Liste2.Items(i).Value) & " "
      End If
    Next
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    <select id="Liste1" size="3" runat="server">
      <option value="o1">Option 1</option>
      <option value="o2">Option 2</option>
      <option value="o3">Option 3</option>
    </select>
    <select id="Liste2" multiple="true" runat="server">
      <option value="o1">Option 1</option>
      <option value="o2">Option 2</option>
      <option value="o3">Option 3</option>
    </select>
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Abbildung 4.17: Die beiden Auswahllisten werden untersucht.

4.4.7 Komplettes Beispiel

Am Ende dieses Abschnitts geben wir Ihnen noch ein komplettes Beispiel, in dem Sie noch einmal alle Formularfeldtypen wiederfinden und bei dem alle Werte ausgegeben werden. Außerdem wird – wie zuvor schon einmal gezeigt – nach der Eingabe der Daten das Formular ausgeblendet, indem seine `Visible`-Eigenschaft auf `False` gesetzt wird. Da es vom theoretischen Aspekt her keine Neuerungen gibt, geht es gleich mit dem Code los.

Listing 4.20: Das komplette Beispiel (htmlausgabe.aspx)

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Versand(ByVal o As Object, ByVal e As EventArgs)
    ausgabe.InnerHtml = "Textfeld: " & Textfeld.Value
    ausgabe.InnerHtml &= "<br />Passwortfeld: " & _
                         Passwortfeld.Value & "<br />"
    ausgabe.InnerHtml &= "Mehrzeiliges Textfeld: " & _
                         Mehrzeilig.Value & "<br />"
    ausgabe.InnerHtml &= "Checkbox: " & Checkbox.Checked
    ausgabe.InnerHtml &= "<br />Radiobutton: " & _
                         Request.Form("Radio")
    ausgabe.InnerHtml &= "<br />Auswahlliste: "
    For i As Integer = 0 To Auswahlliste.Items.Count - 1
      If Auswahlliste.Items(i).Selected Then
        ausgabe.InnerHtml += Auswahlliste.Items(i).Value & " "
      End If
    Next
    form1.Visible = False
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" runat="server">
    Textfeld:
    <input type="text" id="Textfeld" runat="server" />
    <br />
    Passwortfeld:
    <input type="password" id="Passwortfeld" runat="server" />
    <br />
    Mehrzeiliges Textfeld
    <textarea id="Mehrzeilig" wrap="virtual" runat="server" />
    <br />
    Checkbox
    <input type="checkbox" id="Checkbox" value="an" runat="server" />
    <br />
```

```
    Radiobutton
    <input type="radio" name="Radio" id="r1" runat="server" />1
    <input type="radio" name="Radio" id="r2" runat="server" />2
    <br />
    Auswahlliste
    <select id="Auswahlliste" size="3" multiple runat="server">
      <option value="o1">Option 1</option>
      <option value="o2">Option 2</option>
      <option value="o3">Option 3</option>
    </select>
    <br />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

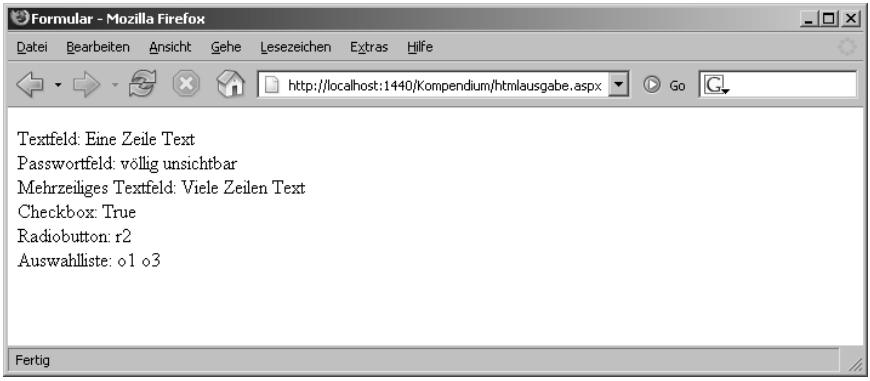


Abbildung 4.18: Die Formulardaten werden ausgegeben und das Formular verschwindet.

4.5 Spezialfall File-Upload

Ein Formularfeld haben wir bis jetzt immer außen vor gelassen – die Rede ist von `<input type="file">`. Damit können Dateien an den Webserver übertragen werden. Bei ASP war der Zugriff auf diese Daten entweder nicht zuverlässig oder musste über Third-Party-Komponenten realisiert werden. Diese Zeiten sind seit dem Erscheinen von ASP.NET 1.0 passé, denn dort ist die Unterstützung für Datei-Uploads integriert.

Zunächst müssen Sie das Formular auf den File-Upload vorbereiten. Dazu müssen Sie das Attribut `enctype` (gibt den Kodierungstyp für die Daten an) auf `"multipart/form-data"` setzen:

```
<form enctype="multipart/form-data" runat="server">
```

Damit werden die Formulardaten nicht wie gewöhnlich als Name-Wert-Paare verschickt (siehe vorheriges Kapitel), sondern als einzelne, MIME-codierte Teile.

Erstellen Sie anschließend ein entsprechendes Formularelement, und vergessen Sie nicht die ID und – ganz wichtig – das `runat="server"`:

```
<input type="file" id="datei" runat="server" />
```

Nach dem Formularversand erhalten Sie über `datei.PostedFile` eine Referenz auf die übertragene Datei. Um genau zu sein, erhalten Sie ein Objekt des Typs `HttpPostedFile`, das unterhalb von `System.Web` angesiedelt ist. Dieses Objekt hat die Methode `SaveAs`, mit der Sie die Datei an einer zu spezifizierenden Stelle ablegen können – ASP.NET benötigt natürlich die entsprechenden Schreibrechte dafür!

```
datei.PostedFile.SaveAs("c:\temp\datei.xxx")
```

Die Klasse `HttpPostedFile` hat zudem noch einige Eigenschaften:

Tabelle 4.2: Die Eigenschaften der Klasse `HttpPostedFile`

Eigenschaft	Beschreibung
`ContentLength`	Dateigröße
`ContentType`	MIME-Typ der Datei
`FileName`	Ursprünglicher Dateiname
`InputStream`	`Stream`-Objekt für die übertragene Datei

Im folgenden Beispiel übertragen wir eines der Listings aus diesem Kapitel an den Webserver und geben die entsprechenden Eigenschaften der Klasse `HttpPostedFile` aus:

Listing 4.21: Die Eigenschaften der übertragenen Datei werden ausgegeben (htmlupload1.aspx).

```
<%@ Page Language="vb" %>
<script runat="server">
Sub Versand(ByVal o As Object, ByVal e As EventArgs)
  Dim d As HttpPostedFile
  d = datei.PostedFile
  ausgabe.InnerHtml = "Gr&ouml;&szlig;e: " & _
                      d.ContentLength & _
                      "<br />MIME-Typ: " & _
                      d.ContentType & _
                      "<br />Dateiname: " & _
                      d.FileName
End Sub
</script>
<html>
<head>
<title>File-Upload</title>
</head>
<body>
<p id="ausgabe" runat="server" />
<form enctype="multipart/form-data" runat="server">
  <input type="file" id="datei" runat="server" />
  <input type="submit" value="Versenden"
         OnServerClick="Versand" runat="server" />
</form>
</body>
</html>
```

Je nach verwendetem Browser erfolgt eine unterschiedliche Ausgabe. Das ist insbesondere anhand der MIME-Typen ersichtlich. Der Internet Explorer beispielsweise verwendet bei *.aspx*-Dateien den MIME-Typ *text/html*, der für HTML-Dateien vorgesehen ist; Mozilla-Browser beispielsweise übermitteln *application/octet-stream*. Besonders problematisch ist es jedoch bei der Eigenschaft `FileName`. Gemäß offizieller Dokumentation enthält diese Eigenschaft den originalen Dateinamen der übertragenen Datei auf dem Client-System, inklusive Pfad. Das trifft allerdings nur beim Microsoft Internet Explorer zu, denn es ist natürlich Sache des Browsers, diese Daten zu übermitteln. Während also der Internet Explorer sehr geschwätzig ist und den kompletten Dateinamen samt Pfad übermittelt, beschränken sich andere Browser auf den bloßen Dateinamen (was auch sinnvoll ist). In Abbildung 4.19 sehen Sie die Ausgabe im Internet Explorer, Abbildung 4.20 zeigt das Skript im Firefox, der auf Mozilla basiert.

TIPP

Sie sehen also: Überprüfen Sie Ihre Skripts unbedingt in unterschiedlichen Browsern!

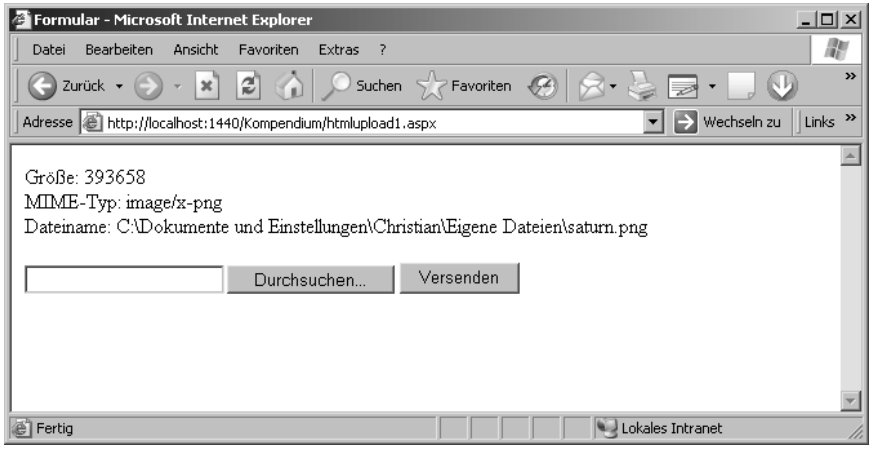


Abbildung 4.19: Der File-Upload im Internet Explorer

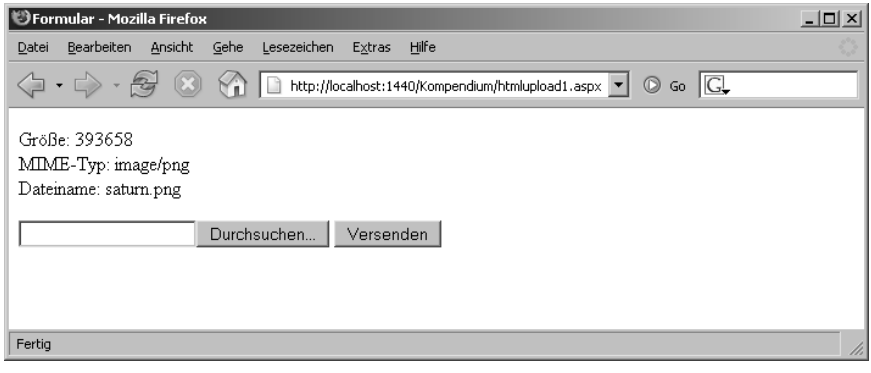


Abbildung 4.20: Der File-Upload im Firefox (lediglich der Dateiname, kein Pfad)

Zum Abschluss dieses Abschnitts ein noch etwas komplexeres Beispiel. Es soll – beispielsweise als Bestandteil eines *Content Management Systems* (CMS) – eine Grafik

auf den Server übertragen und dann direkt angezeigt werden. Dabei gehen wir folgendermaßen vor:

- Der originale Dateiname der übertragenen Datei wird ermittelt. Dazu wird der letzte Backslash im Dateinamen gesucht (falls vorhanden) und danach alle Zeichen im String verwendet.
- Die übertragene Datei wird in ein Verzeichnis *temp* unterhalb des aktuellen Verzeichnisses kopiert. Als Dateiname wird der zuvor ermittelte Name verwendet.
- Schließlich wird die Datei als Grafik in die HTML-Seite eingebunden. Der Dateiname sollte mittlerweile hinlänglich bekannt sein.

Beginnen wir mit der Ermittlung des Dateinamens. Wie erläutert, wird nach dem letzten Backslash im Dateinamen gesucht. Alle folgenden Zeichen gehören dann zum Dateinamen. Wenn der Dateiname von vornherein keinen Backslash enthält (beispielsweise bei Verwendung eines Netscape-Browsers), wird der gesamte Name verwendet:

```
Dim dateiname As String = d.FileName
If dateiname <> "" Then ' überhaupt Upload?
  Dim start As Integer
  start = dateiname.LastIndexOf("\")
  dateiname = dateiname.Substring( _
                start + 1, dateiname.Length - start - 1)
  ' ... Weiterverarbeitung der Datei
End If
```


Der zweite Schritt besteht aus dem Kopieren der Datei:

```
Dim pfad As String
pfad = HttpServerUtility.MapPath("./temp/")
pfad += HttpUtility.UrlEncode(dateiname)
d.SaveAs(pfad)
```

Im letzten Schritt wird die Grafik in die Seite eingebunden – dazu wird der Name vorher noch von Sonderzeichen befreit bzw. diese in das korrekte (URL-)Format gebracht:

```
ausgabe.InnerHtml = "<img src=""temp/" & _
  HttpUtility.UrlEncode(dateiname) & """ />"
```

Hier nun der vollständige Code:

Listing 4.22: Die Datei wird übertragen und ausgegeben (htmlupload2.aspx).


```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Versand(ByVal o As Object, ByVal e As EventArgs)
    Dim d As HttpPostedFile
    d = datei.PostedFile
    Dim dateiname As String
    dateiname = d.FileName
    If dateiname <> "" Then ' überhaupt Upload?
```

```
      Dim start As Integer
      start = dateiname.LastIndexOf("\")
      dateiname = dateiname.Substring( _
        start + 1, dateiname.Length - start - 1)
      Dim pfad As String
      pfad = Server.MapPath("./temp/")
      pfad += dateiname
      d.SaveAs(pfad)
      ausgabe.InnerHtml = "<img src=""temp/" & _
        HttpUtility.UrlEncode(dateiname) & """ />"
    End If
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Formular</title>
</head>
<body>
  <p id="ausgabe" runat="server" />
  <form id="form1" enctype="multipart/form-data" runat="server">
    <input type="file" id="datei" runat="server" />
    <input id="Submit" type="submit" value="Versenden" onserverclick="Versand"
runat="server" />
  </form>
</body>
</html>
```

Abbildung 4.21: Die Grafik wurde hochgeladen und direkt eingebunden.

Bevor Sie diese Applikation auf einer öffentlich zugänglichen Website einsetzen, sollten Sie noch einige Sicherheitsüberprüfungen einbauen, die wir hier aus Platzgründen auslassen mussten. Dazu gehören:

- Überprüfung, ob überhaupt eine Grafik hochgeladen wurde (mindestens Überprüfung der Endung)
- Überprüfung, ob der Dateiname schon existiert (sonst würde ja wahllos überschrieben werden können)
- Überprüfung der Dateigröße (zu große Dateien sollten Sie nicht abspeichern, ansonsten stößt Ihr Webserver vermutlich mittelfristig an seine Grenzen)

Die übertragene Datei wird übrigens nur dann auf dem Webserver abgelegt, wenn Sie sie explizit mit `SaveAs()` *abspeichern; andernfalls löscht der ASP.NET-Prozess die temporäre Datei nach Beendigung des Skripts.*

4.6 Daten im Kopfabschnitt der Seite

Wie in Tabelle 4.1 bereits angedeutet, gibt es im .NET Framework auch HTML Controls, die sich um den Kopfabschnitt (`<head>`) einer Seite kümmern:

- `HTMLHead` steht für den < head > -Abschnitt der Seite an sich
- `HtmlMeta` steht für einen `<meta>`-Tag im `<head>`-Abschnitt der Seite

Der Zugriff erfolgt wieder wie gehabt: Sobald ein `runat="server"` am Element steht und es eine ID besitzt, ist OOP-Zugriff möglich. Das folgende Skript füllt ein (noch leeres) `<meta>`-Tag mit Inhalt:

Listing 4.23: Ein Meta-Element wird dynamisch erzeugt (meta.aspx).

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Page_Load()
    meta1.Name = "date"
    meta1.Content = DateTime.Now.ToString("yyyy-MM-dd hh:mm:ss")
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Meta</title>
  <meta id="meta1" runat="server" />
</head>
<body>
</body>
</html>
```

Das Ergebnis dieses ASP.NET-Skripts enthält unter anderem ein `<meta>`-Element nach folgendem Muster:

```
<meta id="meta1" name="date" content="2006-03-31 04:13:18" />
```

Einige Varianten `<meta>`*-Tags erfordern als Attribut* `http-equiv`*; entsprechend besitzt auch das ASP.NET-Objekt* `HTMLMeta` *die korrespondierende Eigenschaft* `HttpEquiv`.

Der Zugriff auf den `<head>`-Abschnitt einer Seite erfolgt entweder über dessen ID oder über die spezielle Eigenschaft `Page.Header`. Die Hauptanwendung besteht dann darin, auf die Eigenschaft Title zuzugreifen, denn die enthält den Seitentitel (und damit das, was zwischen `<title>` und `</title>` im HTML-Markup steht). Hier ein Beispiel dafür:

Listing 4.24: Der Seitentitel wird dynamisch geändert (head.aspx).

```
<%@ Page Language="VB" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  Sub Page_Load()
    Page.Header.Title = "ASP.NET 2.0"
  End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Head</title>
</head>
<body>
</body>
</html>
```

Hier die Ausgabe von Listing 4.22 (etwas optisch verschönert):

```
<!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><title>
  ASP.NET 2.0
</title></head>
<body>
</body>
</html>
```

ASP.NET hat also automatisch den Inhalt des `<title>`-Elements angepasst. Gerade bei einem Content Management System, bei dem eine Seite unter anderem einen speziellen Titel haben soll, ist diese Zugriffsmöglichkeit sehr hilfreich.

4.7 Fazit

In diesem Kapitel haben Sie zunächst den herkömmlichen Zugriff auf HTTP-Daten, die per GET und POST beim Skript eintreffen, kennen gelernt. Dann haben Sie mit HTML Controls eine Möglichkeit im Einsatz gesehen, einen bequemen Zugriff auf HTML-Formularelemente zu erhalten. Eine Migrierung bestehender HTML-ASP-Formulare auf HTML Controls ist recht schnell erledigt; insbesondere der oft aufwändige Code für die Vorausfüllung kann wegfallen. Deswegen können wir den Einsatz dieser Elemente empfehlen, weil sich allein schon die Entwicklungszeit im Vergleich zu früher verringert. Auch ist es relativ einfach, alte Projekte (etwa in ASP) auf ASP.NET 2.0 zu migrieren.

Wenn Sie jedoch ein Projekt ganz neu von vorne aufziehen, sollten Sie einen Blick in das nächste Kapitel werfen. Dort stellen wir Ihnen eine weitere, sehr mächtige Form serverseitiger ASP.NET-Controls vor.