4 Das Webforms-Programmiermodell

Mit den ASP.NET-Webforms gibt es eine gravierende Änderung im Programmiermodell für serverseitige Webanwendungen. Dieses Kapitel geht ausführlicher auf das ereignisbasierte Webforms-Programmiermodell ein.

4.1 Rückblick

Als Grundlage zum Verständnis des Webforms-Programmiermodells soll hier zunächst kurz an bekannte Programmiermodelle erinnert werden.

Das klassische ASP-Programmiermodell

Generieren von HTML-Code

Im klassischen Webserver-Scripting (ASP und verwandte Plattformen wie PHP und JSP) erzeugt der Programmcode die HTML-Dokumente durch »Hinausschreiben« von HTML-Quellcode, wobei die HTML-Syntax zu beachten ist, wenn man sinnvolle Ergebnisse erhalten will. Die falsche Verwendung von Tags und Attributen führt zu fehlerhaften Bildschirmausgaben. Es gibt dabei keinerlei Syntaxprüfung für den HTML-Code auf dem Server, sodass Fehler im HTML-Quellcode erst durch fehlerhafte Ausgaben im Browser festgestellt werden. Die Erzeugung browserspezifischer Ausgaben ist zwar möglich, macht aber zahlreiche Fallunterscheidungen im Programmcode notwendig.

Server-Tags

Die Ausführung einer Datei im klassischen ASP ist im Wesentlichen linear: Sie beginnt in der ersten Codezeile und erfolgt dann Befehl für Befehl, egal ob es sich um einen HTML-Befehl oder um einen Programmierbefehl in ASP-Tags (`<%...%>` bzw. `<Script runat="server">...</script>`) handelt. Die Linearität wird allenfalls dadurch unterbrochen, dass es Schleifen und Sprünge (Bedingte Ausführung oder Unterprogrammaufrufe) innerhalb der Programmierbefehle gibt.

Eingabeprüfung

Ein nicht unwesentlicher Teil der Programmierung im klassischen ASP beschäftigt sich damit herauszufinden, was der Benutzer auf der aufrufenden Seite eingegeben bzw. angeklickt hat und ob diese Eingaben korrekt sind.

Das ereignisorientierte Desktop-Programmiermodell

Steuerelemente und Ereignisse

Bei der Entwicklung von Desktop-Anwendungen (z.B. Visual Basic, VBA, Visual C++/MFC, Delphi) und bei der clientseitigen Programmierung im Web (Dynamic HTML-Scripting) hat sich ein objektorientiertes, ereignisgesteuertes Programmiermodell durchgesetzt. In diesem Programmiermodell gibt es *Steuerelemente* (engl. *Controls*), die Objekte sind und die *Ereignisse* (engl. *Events*) auslösen, wenn der Benutzer bestimmte Aktionen ausführt. Der Programmierer kann Ereignisbehandlungsroutinen hinterlegen, die (unmittelbar) nach Auslösen des Ereignisses abgearbeitet werden.

Ereignisbehandlung

Durch ein Ereignis in einem Steuerelement wird eine Ereignisbehandlungsroutine aufgerufen, die Änderungen an den anderen Steuerelementen durch Zugriff auf deren Attribute und Methoden vornimmt. Die Reihenfolge der Abarbeitung des Programmcodes ist nicht linear, sondern durch die ausgelösten Aktionen bestimmt. Änderungen an den Steuerelementen werden durch objektorientierte Programmierung (Zugriff auf Attribute, Aufruf von Methoden) ausgeführt.

Ereignisbasierte Programmierung im Web

VI 6.0, WebClasses

Historisch gesehen ist die Idee von serverseitigen Controls und die ereignisbasierte Programmierung im Web nicht neu: Mit den Design-Time-Controls und der zugehörigen Script Library in Visual Interdev 6.0 und den WebClasses in Visual Basic 6.0 hat Microsoft schon zwei ähnliche Ansätze verfolgt. Diese beiden Ansätze waren aber bei weitem nicht so ausgereift wie ASP.NET.

4.2 Webforms und Webcontrols

Webseite als serverseitige Objekthierarchie

Das Webforms-Programmiermodell bildet ein objektorientiertes, ereignisgesteuertes Programmiermodell im Client-Server-Umfeld nach. Es betrachtet nun auch für die serverseitige Web-Programmierung eine Webseite als eine Hierarchie von Objekten (so genannte *Webcontrols*), wobei jedes Objekt Attribute, Methoden und Ereignisse besitzt. Für die Ereignisse können Ereignisbehandlungsroutinen hinterlegt werden.

Umwandlung in HTML-Code

Umwandlung in HTML

Ein Webform wird auf der Serverseite in HTML-Code (und eventuell etwas Client-Script-Code) umgesetzt, d.h. aus jedem Webcontrol werden ein oder mehrere HTML-Tags generiert. Der Client enthält nur HTML-Code, weil alle Webcontrols bereits auf der Serverseite in HTML-Tags umgesetzt werden. Beispielsweise wird aus dem `Textbox` bei der Umwandlung das HTML-Tag `<input type=text>`. Bei der

Umwandlung werden browserspezifische Eigenarten berücksichtigt (dies gilt zumindest für die bei ASP.NET mitgelieferten Webcontrols).

Vergleich mit anderen Technologien

> Vorläufer der ASP.NET-Webcontrols sind die Design Time Controls (DTC) in Visual InterDev 6.0. Die InterDev-DTCs haben sich aufgrund von Schwächen in der Konzeption und Stabilität nicht durchgesetzt. Die Webcontrols sind eine wesentlich bessere Umsetzung dieses Grundgedankens.
>
> Man sollte Webcontrols nicht mit ActiveX-Steuerelementen verwechseln, denn bei der Verwendung von Webforms werden keine ActiveX-Komponenten an den WeFCLient übermittelt.

Unterschied zu Desktop-Anwendungen

Ort der Ereignisbehandlung

Ein wesentlicher Unterschied zur Entwicklung von Desktop-Anwendungen liegt darin, dass das Auslösen des Ereignisses und die Ereignisbehandlung für einige Ereignisse auf zwei verschiedenen Computern (zumindest in zwei verschiedenen Prozessen, wenn Webbrowser und Webserver auf dem gleichen Computer laufen) stattfinden.

Aufbau von Webcontrols

.aspx

Ein Webform ist eine Textdatei mit der Extension *.aspx*. Sie enthält HTML-Code und spezielle Tags für die Webcontrols. Webcontrols unterscheiden sich innerhalb der ASPX-Seite durch drei Merkmale von HTML-Tags:

1. Die Webcontrol-Tags müssen den meisten Regeln der XML-Wohlgeformtheit folgen (die Groß-/Kleinschreibung ist jedoch egal).
2. Sie beginnen meist mit dem Namespace `asp`.
3. Sie haben als Attribut `runat="server"`.

Beispiele für Webcontrols

<form>-Tag

Die folgende Tabelle zeigt Beispiele für Webcontrols. Alle Webcontrols eines Webforms sind wie HTML-Steuerelemente in ein `<form>`-Tag eingebettet – allerdings besitzt auch das `<form>`-Tag das Zusatzattribut `runat="server"`.

Formular	`<form id="Form1" method="post" runat="server">`
	`...`
	`</form>`

Tabelle 4.1: Beispiele für Webcontrols

Texteingabefeld	<asp:TextBox id="F_Name" runat="server" Width="202px"> Schwichtenberg </asp:TextBox>
Liste von Options-schaltflächen	<asp:RadioButtonList id="RadioButtonList1" runat="server" RepeatDirection="Horizontal"> <asp:ListItem Value="m">Männlich </asp:ListItem> <asp:ListItem Value="W" Selected="True">Weiblich </asp:ListItem></asp:RadioButtonList>
Drop-Down-Menü	<asp:DropDownList id="F_Beruf" runat="server" Width="201px"> <asp:ListItem Value="Student">Student </asp:ListItem> <asp:ListItem Value="Professor">Professor </asp:ListItem> <asp:ListItem Value="WissenschaftlicherMitarbei ter" Selected="True">WissenschaftlicherMitarbei ter</asp:ListItem> </asp:DropDownList>
Kontrollkästchen	<asp:CheckBox id="CheckBox1" runat="server" Text="Für Eintragung Rückbestätigung per E-Mail anfordern."> </asp:CheckBox>
Schaltfläche	<asp:Button id="B_Eintragen" runat="server" Text="Eintragen"></asp:Button>

Tabelle 4.1: Beispiele für Webcontrols (Fortsetzung)

4.3 Ereignisbehandlung in Webforms

Ereignisse auf dem Server

Genau wie bei Formularen in VB5/6 oder VBA gibt es zu jedem Webcontrol verschiedene Ereignisse, für die Ereignisbehandlungsroutinen hinterlegt werden können. Die Webcontrols bieten typische Ereignisse an, die sinnvoll auf dem Server behandelt werden können, weil sie entweder auf dem Server stattfinden (z.B. »vor dem Generieren eines Elements«, »nach dem Generieren«) oder weil sie vom Server aufgrund der übermittelten Parameter bemerkt werden können. Das sind im Wesentlichen das Ändern eines Steuerelement-Wertes und der Klick auf ein Steuerelement.

Feststellen der Ereignisse

Die Ereignisbehandlungsroutinen werden nicht sofort, sondern erst nach einem Roundtrip zum Server ausgeführt. Genau genommen bekommt der Server gar keine echte Ereignismeldung, sondern vergleicht die Inhalte der Steuerelemente mit den Inhalten bei der letzten Übermittlung (die im *Viewstate* gespeichert wurden, vgl. Kapitel 5.2). Entsprechend der festgestellten Änderungen feuert ASP.NETASP.NET die zugehörigen Ereignisbehandlungsroutinen.

Daher kann man das Webforms-Programmiermodell auch als *pseudo-ereignisbasiert* bezeichnen, weil es keinen wirklichen direkten Zusammenhang zwischen Ereignis und Ereignisbehandlung gibt. Der Entwickler bekommt aber so gut wie gar nichts mehr davon mit, dass ein HTTP-Request zwischen dem Klick auf einen Button und seiner Ereignisbehandlungsroutine liegt.

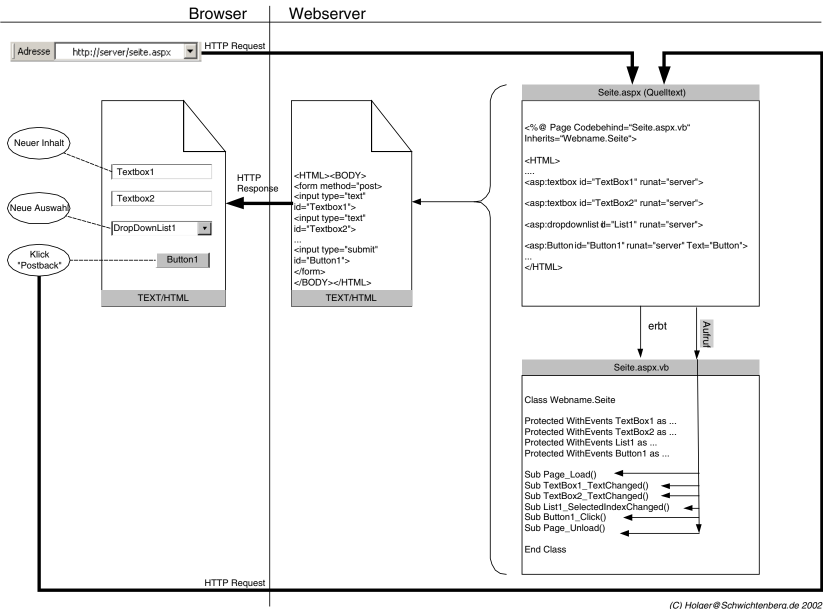

Abbildung 4.1: Serverseitige Ereignisbehandlung in ASP.NET (hier mit Code-Behind-Datei, siehe Kapitel 4.4)

Arten von Ereignissen

Postback

Im normalen HTML löst nur der Klick auf einen Link oder einen »Submit«-Button eine erneute HTTP-Anfrage (*Postback*) aus, nicht aber die Änderung eines Wertes in einem Eingabe-Steuerelement wie einem Textfeld oder einer Checkbox. Dies ist auch das Standardverhalten bei Webcontrols. Erst wenn ein Button gedrückt wurde, wird eine HTTP-Anfrage an den Server gestellt. Und erst dann werden dort neben dem `Click`-Ereignis für den Button auch alle Ereignisse für Zustandsänderungen in Eingabe-Steuerelementen ausgeführt.

Man kann die Webcontrol-Ereignisse in drei Gruppen einordnen:

Server-Ereignisse

1. Ereignisse, die nur auf dem Server stattfinden und die keine Beziehung zum Client haben. Dazu gehört zum Beispiel das `Load()`-Ereignis, das bei der Verarbeitung der ASPX-Seite auf dem Server aufgerufen wird.

Non-Postback-Events

2. Ereignisse, die auf dem Client stattfinden, die aber (im Standard) nicht zu einer sofortigen Rückfrage (Roundtrip) zum Server führen. Dies sind alle Änderungen in Eingabe-Steuerelementen (z.B. `TextChanged()`, `SelectedIndexChanged()`). Der Server erfährt von diesen Ereignissen erst beim nächsten Roundtrip durch einen Vergleich der alten und neuen Werte. Für jede Wertänderung feuert er dann serverseitig ein Ereignis. Diese Ereignisse heißen **Non-Postback-Ereignisse**.

Postback-Events

3. Ereignisse, die auf dem Client stattfinden und die sofort zum Server weitergeleitet werden (**Postback-Ereignisse**). Dies ist in HTML im Standard nur der Klick auf einen »Submit«-Button (`<input type="submit"...>`).

Die Ereignisse der Gruppe 2 können aber durch Einsatz von JavaScript zu diesem Verhalten gebracht werden. Man kann also ein Non-Postback-Ereignis zu einem Postback-Ereignis machen. Dies wird weiter unten erläutert.

Ereignis	Typ	Beschreibung
`Load()`	Reines Serverereignis	Ereignis vor der Generierung des HTML-Codes für ein Steuerelement
`UnLoad()`	Reines Serverereignis	Ereignis nach der Generierung des HTML-Codes für ein Steuerelement
`Click()`	Postback-Ereignis	Klick auf einen Button
`TextChanged()`	Non-Postback-Ereignis	Beenden der Eingabe in ein Textfeld
`SelectedIndex-Changed()`	Non-Postback-Ereignis	Ändern der Auswahl in einem Drop-Down-Feld
`CheckedChanged()`	Non-Postback-Ereignis	Änderung des Zustands einer Checkbox

Tabelle 4.2: Wichtige Webcontrol-Ereignisse als Beispiel; alle Non-Postback-Ereignisse können optional zu Postback-Ereignissen werden.

Ereignisparameter

Quelle und Details

Die Webcontrol-Ereignisse besitzen jeweils zwei Parameter: Im ersten Parameter wird ein Zeiger auf das Objekt übergeben, das das Ereignis ausgelöst hat. So ist es möglich, eine Ereignisbehandlungsroutine für mehrere Ereignisse zu schreiben. Im zweiten Parameter werden (bei einigen Steuerelementen) Details zum Ereignis übergeben.

```
Sub B_Eintragen_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs)
```

Listing 4.1: Beispiel für die Ereignis-Parameter bei einem Click()-Ereignis eines Button-Controls

Serverseitige vs. clientseitige Ereignisse

Webcontrol-Ereignisse versus DHTML-Ereignisse

Die Anzahl der Webcontrol-Ereignisse ist aber typischerweise deutlich geringer als die der clientseitigen DHTML-Ereignisse. Dies ist auch klar, denn Ereignisse wie `MouseDown()`, `KeyPressed()` oder `Enter()` auf dem Server zu behandeln, macht angesichts der Zeit, die ein Roundtrip zum Server benötigt, keinen Sinn. Derartige Ereignisse müssen weiterhin durch Dynamic HTML (DHTML) auf dem Client behandelt werden. Überhaupt würde der Server von diesen Ereignissen nur durch den Einsatz von zusätzlichen Client-Scripts erfahren, denn diese Ereignisse führen ja nicht notwendigerweise zu einer Änderung des Inhalts, die der Server bemerken würde.

Wahlweise können einige Webcontrols (z.B. Validation Controls) auch in clientseitigen Code umgesetzt werden, siehe Kapitel 5.5.

Sofortige Ereignisweiterleitung

AutoPostBack

Bei vielen Webcontrols kann man ein Non-Postback-Ereignis zu einem Postback-Ereignis machen. Vom Entwickler kann über das Webcontrol-Attribut `AutoPostBack` definiert werden, ob eine Zustandsänderung in einem Eingabesteuerelement zum sofortigen Postback führt.

Da ein Postback in HTML nur für »`Submit`«-Buttons, nicht aber für alle anderen Steuerelemente vorgesehen ist, muss ASP.NET ein Stück JavaScript-Code in die Seite einfügen, welches das Formular zum Postback zwingt.

```
<!--  function __doPostBack(eventTarget, eventArgument) {
var theform = document.Form1;
theform.__EVENTTARGET.value = eventTarget;
theform.__EVENTARGUMENT.value = eventArgument;
theform.submit();
  }
// -->
```

Das generierte HTML-Tag erhält dann diese Funktion als Ereignisbehandlungsroutine für das passende DHTML-Ereignis. Im Fall der Textbox entspricht ein `onchange()` in DHTML einem `TextChanged()` in ASP.NET.

```
<input name="TextBox1" type="text" id="TextBox1"
onchange="__doPostBack('TextBox1','')" language="javascript" />
```

Wenn im Browser JavaScript deaktiviert ist, wird ein solcher manueller Postback nicht funktionieren. Sie sollten also keine Webforms erzeugen, die nur mit einem manuellen Postback verlassen werden können.

4.4 Seitenmodelle

Es gibt zwei Modelle für die Speicherung von ASP.NET-Seiten:

1. Modell mit einer Datei (*Single-File-Modell*)
2. Modell mit zwei Dateien (*Code-Behind-Modell*)

Single-File-Modell

Vermischung von Layout und Programmcode

Beim Single-File-Modell gibt es nur eine ASPX-Datei, in der HTML-Code und serverseitiger Programmcode zusammen liegen. Dies ist das aus dem klassischen ASP bekannte Seitenmodell. Der Hauptnachteil dieses Seitenmodells ist, dass durch die Vermischung von Layout und Programmcode in einer Datei die Zusammenarbeit zwischen Webdesigner und Webentwickler erschwert ist.

Beim Code-Behind-Modell, das in Kapitel 4.4 vorgestellt wird, besteht die Möglichkeit, den Programmcode in eine eigene Datei auszulagern.

Einbettung von Code

In der ASPX-Seite werden Codeblöcke wie im klassischen ASP durch die folgenden zwei alternativen Begrenzer kenntlich gemacht.

Kurzform	`<% ... %>`
Ausführliche Form	`<script language="VB" runat="server"> ... </script>`

Tabelle 1:

Code-Begrenzer

Während in ASP diese Begrenzer äquivalent waren, gibt es nun einen Unterschied:

1. Die Begrenzer `<% %>` dürfen keine Deklaration von Unterroutinen enthalten, sondern nur noch einzelne »freistehende« Befehle.

2. Im Umkehrschluss darf der Begrenzer `<script> </script>` nur noch Unterroutinen, aber keine freistehenden Befehle mehr abarbeiten.

Festlegung der Programmiersprache

Es kann in jeder ASPX-Seite nur eine einzige Programmiersprache verwendet werden. Diese wird entweder durch das `language`-Attribut in der `@Page`-Direktive oder aber durch das gleichnamige Attribut in dem `<script>`-Tag festgelegt. Wenn die `@Page`-Direktive eine Sprache festlegt, dann ist die Angabe des `language`-Attributs in dem `<script>`-Tag optional, denn hier darf sowieso keine andere Sprache angegeben werden.

Gültige Sprachen

Die Zuordnung Sprachkürzel und Sprach-Compiler geschieht über eine Konfigurationseinstellung, die in der globalen Konfigurationsdatei *machine.config* hinterlegt ist. Im Standard sind dort die drei mit dem .NET Framework ausgelieferten Sprachen Visual Basic .NET, C# und JScript .NET definiert. Für jede der drei Sprachen sind verschiedene alternative Abkürzungen definiert. Die Standardsprache, die verwendet wird, wenn das `language`-Attribut fehlt, ist Visual Basic .NET.

```
<system.web>
...
<compilation debug="false" explicit="true" defaultLanguage="vb">
<compilers>
<compiler language="c#;cs;csharp" extension=".cs"
type="Microsoft.CSharp.CSharpCodeProvider, System, Version=1.0.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" warningLevel="1" />
<compiler language="vb;vbs;visualbasic;vbscript" extension=".vb"
type="Microsoft.VisualBasic.VBCodeProvider, System,
Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
/>
<compiler language="js;jscript;javascript" extension=".js"
type="Microsoft.JScript.JScriptCodeProvider, Microsoft.JScript,
Version=7.0.3300.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
/>
</compilers>
...
</compilation>
...
</system.web>
```

Listing 4.2: Ausschnitt aus machine.config

Keine Skriptsprachen

Bitte beachten Sie, dass als Sprachen nur noch .NET-Sprachen erlaubt sind. Die COM-basierten Active Scripting-Sprachen (z.B. VBScript und JScript) sind jetzt nicht mehr verwendbar. Zwar definiert die *machine.config* »VBScript« und »JScript« als Sprachkürzel, erwartet wird aber in beiden Fällen die Syntax des jeweiligen .NET-Pendants.

Kompilierung

Erstellung einer Assembly

ASPX-Webseiten werden nicht interpretiert, sondern einmalig – beim ersten Aufruf – kompiliert und in einer Assembly (*.DLL*) im Dateisystem gespeichert. Für jede ASPX-Datei wird eine Assembly in der Microsoft Intermediation Language (MSIL) erzeugt. ASP.NET verwendet dann bei jeder Anfrage diese Assembly – solange, bis sich die ASPX-Datei geändert hat und daher eine Neukompilierung erforderlich ist. ASP.NET vergleicht also bei jedem Seitenabruf zunächst den Erstellungszeitpunkt der ASPX-Datei mit dem der zugehörigen Assembly.

Die MSIL-Assembly wird dann – wie in .NET üblich – während der Ausführung Unterroutine für Unterroutine von einem Just-in-Time-Compiler in Native Code übersetzt (siehe Abbildung 4.2).

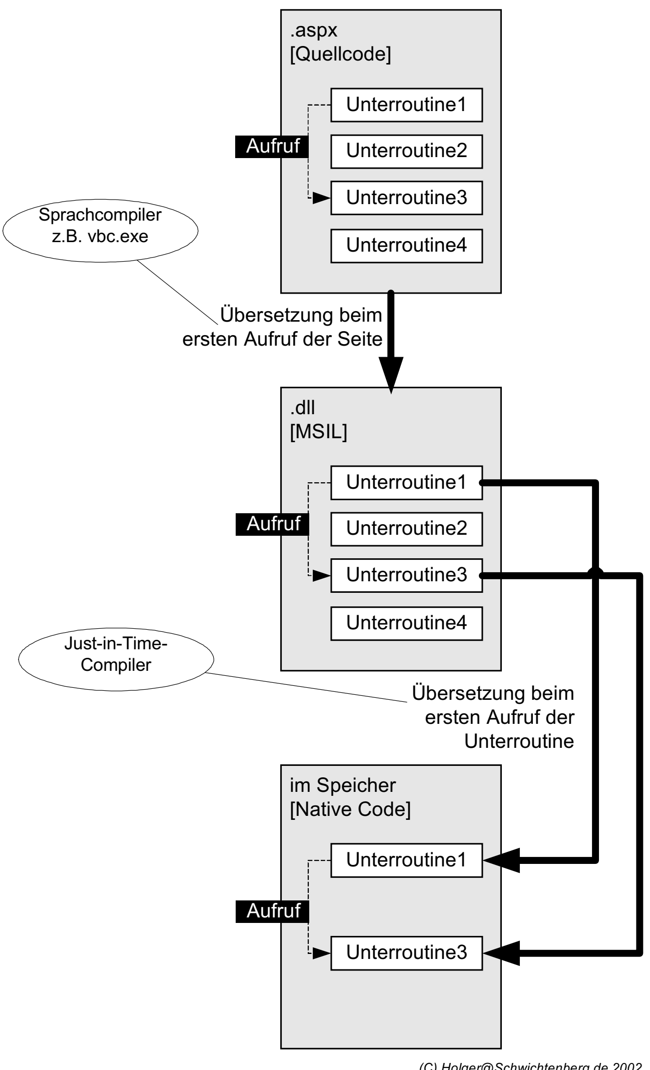

Abbildung 4.2: Wege des Codes in ASP.NET im Single-File-Modell

Umwandlung in Quellcode

Ganz genau sieht der Prozess so aus: ASP.NET erzeugt aus der ASPX-Datei zunächst Programm-Quellcode, d.h. auch die HTML-Tags und die statischen Inhalte werden in Programmcode umgewandelt. Der in der ASPX-Datei enthaltene serverseitige Programmcode wird 1:1 in die neue Quellcode-Datei übernommen. Es entsteht eine Klasse in dem Namespace ASP, die so heißt wie die ASPX-Seite, wobei der Punkt in einen Unterstrich umgewandelt wird. Aus *Schnellstart_sf.aspx* wird also die Klasse `ASP.Schnellstart_sf_aspx`. Diese Klasse erbt direkt von der FCL-Klasse `System.Web.UI.Page`.

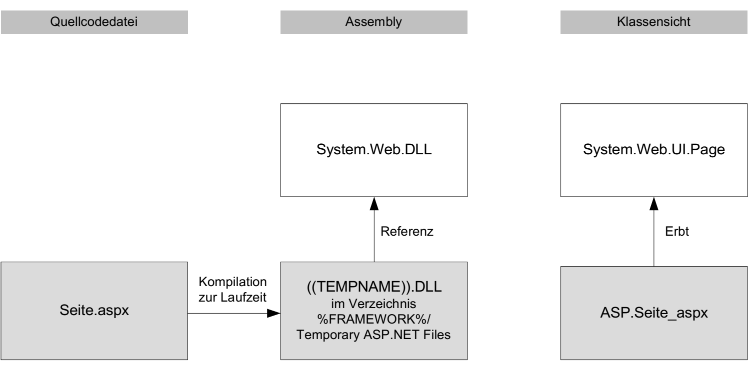

Abbildung 4.3: Dateien und Klassen im Single-File-Modell

Kommandozeilen-Compiler

Als Sprache wird bei der automatisch generierten Klasse zwangsläufig die Sprache verwendet, die in der `@Page`-Direktive im Attribut `language` angegeben wurde, denn es ist nicht möglich, zwei Sprachen innerhalb einer Quellcode-Datei zu mischen. Diese Quellcode-Datei wird dann mit dem entsprechenden Sprach-Compiler in MSIL-Code kompiliert.

> Da die Compiler-Klassen in der FCL auf die Kommandozeilen-Compiler zurückgreifen, kann ASP.NET nicht funktionieren, wenn der entsprechende Kommandozeilen-Compiler nicht im *%Framework%*-Verzeichnis vorhanden ist.

```
Option Strict Off
Option Explicit On

Imports ASP
Imports Microsoft.VisualBasic
Imports System
...

Namespace ASP
    <System.Runtime.CompilerServices.CompilerGlobalScopeAttribute()>
...
    Public Class Schnellstart_sf_aspx
   Inherits System.Web.UI.Page
   Implements System.Web.SessionState.IRequiresSessionState
   Private Shared __autoHandlers As Integer
#ExternalSource("f:\web\wfbuch\schnellstart\schnellstart_sf.aspx",12)
   Protected L_Schritt1 As System.Web.UI.WebControls.Label
      #End ExternalSource
...
   Private Sub __BuildControlTree(ByVal __ctrl As
System.Web.UI.Control)
```

```
  Dim __parser As System.Web.UI.IParserAccessor =
CType(__ctrl,System.Web.UI.IParserAccessor)
 #ExternalSource("f:\web\wfbuch\schnellstart\schnellstart_sf.aspx",1)
  __parser.AddParsedSubObject(New
System.Web.UI.LiteralControl(""&Microsoft.VisualBasic.ChrW(13)&Micro
soft.VisualBasic.ChrW(10)&Microsoft.VisualBasic.ChrW(13)&Microsoft.
VisualBasic.ChrW(10)&"<!DOCTYPE HTML PUBLIC ""-//W3C//DTD HTML 4.0
Transitional//
EN"">"&Microsoft.VisualBasic.ChrW(13)&Microsoft.VisualBasic.ChrW(10)&"
<HTML>"&Microsoft.VisualBasic.ChrW(13)&Microsoft.VisualBasic.ChrW(10)&
Microsoft.VisualBasic.ChrW(9)&"<HEA"& _
"D>"&Microsoft.VisualBasic.ChrW(13)&Microsoft.VisualBasic.ChrW(10)&
Microsoft.VisualBasic.ChrW(9)&Microsoft.VisualBasic.ChrW(9)&"<title>
Erstes Beispiel</
title>"&Microsoft.VisualBasic.ChrW(13)&Microsoft.VisualBasic.ChrW(10)&
Microsoft.VisualBasic.ChrW(9)&"</
HEAD>"&Microsoft.VisualBasic.ChrW(13)&Microsoft.VisualBasic.ChrW(10)&
Microsoft.VisualBasic.ChrW(9)&"<body>"&Microsoft.VisualBasic.ChrW(13)&
Microsoft.VisualBasic.ChrW(10)&Microsoft.VisualBasic.ChrW(9)&Microsoft
.VisualBasic.ChrW(9)))
  #End ExternalSource
 #ExternalSource("f:\web\wfbuch\schnellstart\schnellstart_sf.aspx",1)
  Me.__BuildControlForm1

  #End ExternalSource

#ExternalSource("f:\web\wfbuch\schnellstart\schnellstart_sf.aspx",1)
  __parser.AddParsedSubObject(Me.Form1)

  #End ExternalSource...
```

Listing 4.3: Ausschnitte aus der generierten Quellcode-Datei

ClassName

Für die aus der ASPX-Datei erzeugte Klasse kann auch ein expliziter Name vergeben werden. Dazu ist in der `@Page`-Direktive das Attribut `ClassName` anzugeben.

```
<%@ Page ... Language="vb" ClassName="su" ... %>
```

Abbildung 4.4: Ansicht der generierten Assembly im .NET-Disassembler (ildasm.exe)

Speicherung der temporären Dateien

Komplexe Verzeichnisstruktur

ASP.NET speichert die generierte Quellcode-Datei und die daraus erzeugte Assembly (*.dll*) im Dateisystem ab. Unterhalb von *%WINDIR%\Microsoft.NET\Framework\v1.0.3705\Temporary ASP.NET Files* wird für jedes virtuelle Verzeichnis des IIS (sofern es bereits verwendet wurde) ein Unterverzeichnis angelegt. Von dort aus vergibt ASP.NET die Namen selbst, zumal alle temporären Dateien in einem einzigen Verzeichnis gespeichert werden. Die Verzeichnisstruktur der Webanwendung wird nicht übernommen. Hätten die temporären Dateien die Namen der ASPX-Dateien, könnte es zu Namenskonflikten kommen, denn innerhalb einer Webanwendung kann ein Dateiname in verschiedenen Unterverzeichnissen beliebig oft vorkommen. Die temporären Dateien bekommen Namen, die aus einem 8-stelligen Zeichencode bestehen.

> Sie können das Verzeichnis über das folgende statische Attribut ermitteln: `HttpRuntime.CodegenDir`.

Weitere Dateien

Wie Sie der folgenden Bildschirmkopie entnehmen können, erzeugt ASP.NET noch mehr Dateien:

1. Die Datei mit der Extension *.cmdline* enthält die kompletten Optionen, mit denen der Compiler aufgerufen werden muss, um die Quellcode-Datei richtig zu übersetzen.
2. Die Datei mit der Extension *.out* enthält die Ausgabe des Kommandozeilen-Compilers, die man im Fehlerfall auch innerhalb der Fehlerseite einsehen kann.
3. Die Datei mit der Extension *.pdb* enthält Debug-Informationen, die beim Einsatz eines .NET-Debuggers benötigt werden.

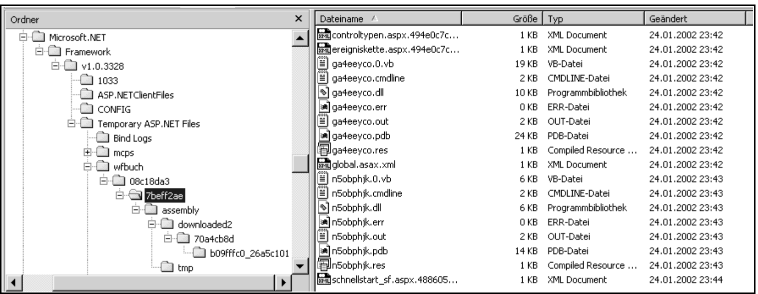

Abbildung 4.5: Einsatz des temporären Verzeichnisses von ASP.NET

Auslagern von Code

Dateien einbinden

Das Single-File-Modell hat seinen Namen nicht ganz zu Recht, denn auch hier gibt es die Möglichkeit, das Programm in eine andere Datei auszulagern. Das `<script>`-Tag unterstützt mit dem Attribut `src` den Verweis auf eine andere Datei, die beim Parsen hinzugeladen wird. In diesem Fall muss das `<script>`-Tag einen leeren Inhalt haben.

```
<script runat="server" src="Dateiname.vb" />
```

Dies sind aber noch keine echten Code-Behind-Dateien, wie sie im nächsten Kapitel erläutert werden.

Ereignisbindung

Ereignisnamen

Ein zentrales Thema ist die Bindung zwischen einem Webcontrol-Ereignis und der Ereignisbehandlungsroutine. In dem Tag eines Webcontrols gibt es daher Attribute, die wie die Ereignisse des Webcontrols heißen, zusätzlich mit der Vorsilbe »on« (z.B. `onclick` für `Click()` und `onload` für `Load()`). Diese Attribute dienen der Festlegung, welche Routine als Ereignisbehandlungsroutine verwendet werden soll.

Der folgende Kasten zeigt die Ereignisbindung für das `Click()`-Ereignis eines Buttons. Im Single-File-Modell, also wenn die Ereignisbehand-

lungsroutine in der gleichen Datei wie das Steuerelement-Tag steht, ist es egal, wie zugänglich (`Public`, `Private` oder `Protected`) die Routine ist.

Steuerelement-Tag	Ereignisbehandlungsroutine
`<asp:button id="B_OK" onclick="B_OK_Geklickt" runat="server" Text="OK"> </asp:button>`	`Sub B_OK_Geklickt (ByVal sender As System.Object, ByVal e As System.EventArgs)`

Tabelle 2:

Ereignisbindung mit Namenskonvention

AutoEventWireUp

Es gibt noch eine alternative Ereignisbindung, die auf der Namensgebung basiert und durch das Attribut `AutoEventWireUp="True"` in der `@Page`-Direktive gesetzt wird. Dabei werden alle Ereignisbehandlungsroutinen ausgeführt, deren Name der Konvention

```
OBJEKTVARIABLE_EREIGNISNAME(PASSENDE_PARAMETERLISTE)
```

entspricht. Dies ist besonders sinnvoll für die Bindung des `Page_Load()`-Ereignisses.

```
<%@ Page Language="vb" AutoEventWireup="true"%>
...
<script runat="server">
Sub Page_Load(Sender As Object, E As EventArgs)
...
End Sub
</script>
```

Listing 4.4: Bindung von Page_Load() wie AutoEventWireup

Code-Behind-Modell

Zwei Dateien

Im Code-Behind-Seitenmodell sind HTML-Code und Programmcode grundsätzlich in zwei Dateien getrennt: Die ASPX-Datei enthält den HTML-Code und eine getrennte Datei den Programmcode. Diese Programmcode-Datei hat nach einer Konvention den gleichen Namen mit zusätzlicher Dateiextension, die die Sprache festlegt.

Beispiel: Die in VB.NET geschriebene Code-Behind-Datei für die ASPX-Datei *cb.aspx* sollte *cb.aspx.vb* heißen.

Zwar ist es möglich, bei einer existierenden Code-Behind-Datei zusätzlich auch noch Programmcode innerhalb der ASPX-Seite zu speichern. Programmcode kann also wahlweise in ASPX-Dateien (wie bisher in .asp-Dateien) oder in die zugehörige Code-Behind-Datei abgelegt wer-

den. Code in der ASPX-Seite ist aber nicht erwünscht (vgl. oben, Single File Modell).

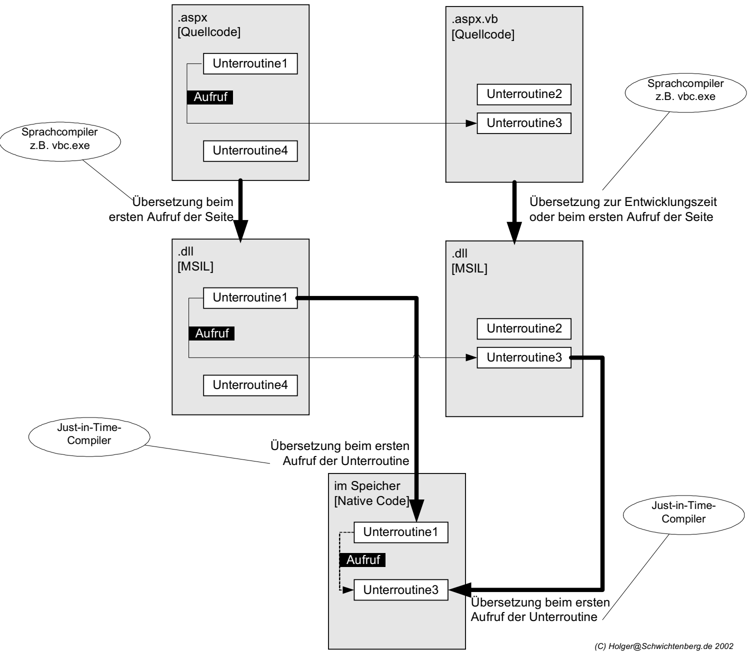

Abbildung 4.6: Wege des Programmcodes in ASP.NET im Code-Behind-Modell

Vererbung

ASPX-Klasse erbt von Code-Behind-Klasse

Das Code-Behind-Modell basiert auf Vererbung: In der Code-Behind-Datei wird eine Klasse implementiert, die von der FCL-Klasse `System.Web.UI.Page` erbt. Diese Klasse hat einen beliebigen Namen, üblich ist aber die Benennung `WEBNAME.SEITENNAME`. Diese Klasse ist wiederum Oberklasse für die ASPX-Seite. Genauer gesagt: Diese Klasse ist Oberklasse für die Webseiten-Klasse (`ASP.SEITENNAME_aspx`), die ASP.NET aus der ASPX-Seite dynamisch generiert.

```
Public Class Schnellstart_CB
  Inherits System.Web.UI.Page
    ...
```

Listing 4.5: Klassendeklaration in der Code-Behind-Datei [Schnellstart_CB.aspx.vb]

Erben von der Code-Behind-Klasse

Im Single-File-Modell erbt die Webseiten-Klasse (also die Klasse `ASP.SEITENNAME_aspx`) direkt von `System.Web.UI.Page`. Im Code-Behind-Modell erbt sie von der Code-Behind-Klasse. Diese Vererbungsbezie-

hung muss man in der ASPX-Seite in der `@Page`-Direktive spezifizieren mit dem Attribut `Inherits`, das den vollständigen Namen der Klasse (also inklusive Namespace) enthalten muss. Dieser Name ist **case-sensitive**.

Abbildung 4.7: Ansicht der von Visual Studio .NET aus den verschiedenen Code-Behind-Dateien erzeugten DLL im IL-Disassembler (ildasm.exe)

```
<%@ Page Language="vb"
Codebehind="Schnellstart_CB.aspx.vb"
Inherits="wf.Schnellstart_CB"%>
```

Listing 4.6: @Page-Direktive in der ASPX-Seite im Code-Behind-Modell [Schnellstart_CB.aspx]

```
Public Class Schnellstart_CB_aspx
  Inherits wf.Schnellstart_CB
...
```

Listing 4.7: Klassendeklaration der von ASP.NET beim ersten Aufruf der ASPX-Seite automatisch erzeugten Webseiten-Klasse

Abbildung 4.8: In der temporären Webseiten-Assembly spiegelt sich die Vererbungsbeziehung wider.

Kompilierung der Code-Behind-Datei

Dynamische Assembly

Die Code-Behind-Datei kann wahlweise zur Entwicklungszeit oder dynamisch zur Laufzeit zu einer Assembly kompiliert werden.

Eine **zur Laufzeit** kompilierte Code-Behind-Datei behandelt ASP.NET wie eine ASPX-Datei und legt die beim ersten Aufruf erzeugte Assembly unterhalb des Verzeichnisses *%WINDIR%\Microsoft.NET\Frame-*

work\v1.0.3705\Temporary ASP.NET Files ab. ASP.NET erzeugt in diesem Fall für jede Code-Behind-Datei genau eine Assembly.

Statische Assembly

Eine **vorkompilierte** DLL muss man im */bin*-Verzeichnis unterhalb des Wurzelverzeichnisses der Webanwendung speichern, weil ASP.NET im Standard nur dort danach sucht. Der Name der DLL ist beliebig, da ASP.NET so voreingestellt ist, dass alle DLLs im */bin*-Verzeichnis hinzugebunden werden. Man hat daher auch die Wahl, für jede einzelne Code-Behind-Datei eine einzelne DLL zu erzeugen oder mehrere Code-Behind-Dateien zu einer DLL zusammenzufassen. Die folgende Anweisung zeigt den Befehl zur Übersetzung einer Code-Behind-Datei mit dem Kommandozeilen-Compiler. Mindestens die *System.dll* und die *System.Web.dll* müssen referenziert werden.

```
vbc.exe cb.aspx.vb /t:library /out:..\bin\hs.dll
/r:System.Web.dll /r:System.dll
```

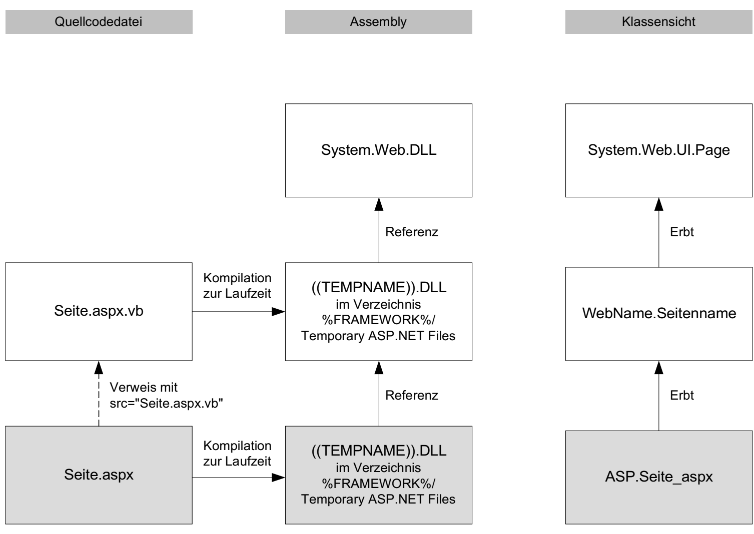

Abbildung 4.9: Laufzeitkompilierung der Code-Behind-Datei

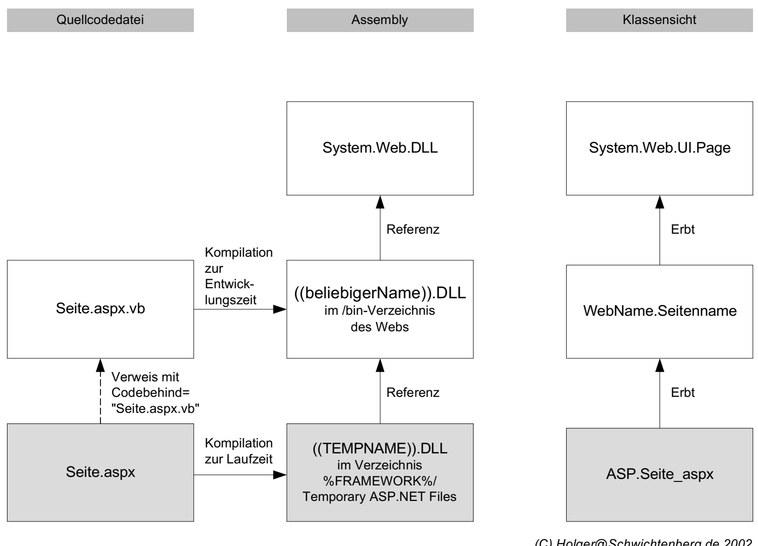

Abbildung 4.10: Entwicklungszeitkompilierung der Code-Behind-Datei

> Visual Studio .NET kompiliert alle Code-Behind-Dateien in eine einzige DLL zusammen, die den Namen der Webanwendung trägt: *WEBNAME.dll*.

Kompilierung der ASPX-Datei

Kompilierung der ASPX-Datei

Die ASPX-Datei wird genau wie im Single-File-Modell erst zur Laufzeit beim ersten Aufruf kompiliert und in Form einer Assembly unterhalb des Verzeichnisses *%WINDIR%\Microsoft.NET\Framework\v1.0.3705\Temporary ASP.NET Files* gespeichert. Bezüglich der Speicherung besteht kein Unterschied, bezüglich der Vererbung schon.

Festlegung der Programmiersprache

Die Sprache der ASPX-Seite wird auch im Code-Behind-Modell nach den gleichen Regeln wie im Single-File-Modell festgelegt. Die verwendete Programmiersprache in der Code-Behind-Datei wird je nach Vorgehensweise bei der Kompilierung bestimmt:

Sprache der Code-Behind-Datei

1. Im Fall der Einzelkompilierung zur Entwicklungszeit per Kommandozeilen-Compiler bestimmt der verwendete Compiler die Sprache der Code-Behind-Datei. Die Dateiextension ist irrelevant und es ist möglich, jede Code-Behind-Datei in einer anderen Sprache zu schreiben.

2. Bei der Verwendung von Visual Studio .NET zur Entwicklungszeitkompilierung ist die Sprache durch das Anlegen des Projekts für alle Code-Behind-Dateien in einem Projekt einheitlich festgelegt, weil Visual Studio .NET eine einzige DLL erzeugt und ein .NET-Modul immer genau in einer Sprache geschrieben sein muss.
3. Nur bei der Laufzeitkompilierung legt die Dateiextension den von ASP.NET automatisch zu verwendenden Compiler fest.

Es ist aber empfehlenswert, in allen Fällen die Dateiextension passend zur verwendeten Sprache zu wählen.

Suchen der Implementierung

Zur Festlegung, wo die ASPX-Seite die Implementierung der Code-Behind-Klasse findet, gibt es zwei Alternativen in der `@Page`-Direktive der ASPX-Datei:

1. `src="DATEINAME.aspx.vb"`

   src

   Mit dem `src`-Attribut wird festgelegt, dass die Code-Behind-Klasse zur Laufzeit dynamisch kompiliert werden soll. `src` verweist auf eine Quellcode-Datei.

2. `Codebehind="DATEINAME.aspx.vb"` oder ohne Angabe

   Codebehind

   Wenn nicht `src` angegeben wird, wird ASP.NET damit beauftragt, die Code-Behind-Klasse in der vorkompilierten Assembly (*WEBNAME.dll*) zu suchen. Die Quellcode-Dateien müssen dann nicht mehr im Verzeichnis der Webanwendung liegen. Die Angabe `Codebehind` wird nur von Visual Studio .NET verwendet, um in der Entwicklungsumgebung die zugehörige Code-Behind-Datei zu lokalisieren. Wenn man nicht VS.NET einsetzt, kann man darauf verzichten.

Wünschenswert wäre eine Fallback-Strategie: Greife nur auf die *WEBNAME.dll* zurück, wenn die Quellcode-Datei nicht existiert (oder umgekehrt). Leider bietet ASP.NET dies **nicht** an, denn wenn das `src`-Attribut angegeben ist, sucht ASP.NET die Code-Behind-Klasse **immer** an dem nach `src` angegebenen Ort. Ist die Datei dort nicht vorhanden oder die Code-Behind-Klasse dort nicht implementiert, gibt es eine Fehlermeldung. ASP.NET bindet zwar die im */bin*-Verzeichnis enthaltenen DLLs immer mit ein, jedoch hilft das nichts: Sind nämlich in den in */bin* gespeicherten DLLs und in der Quellcode-Datei die gleichen Klassen definiert, streikt der Compiler.

Anforderungen an die Code-Behind-Klasse

An die Klasse in der Code-Behind-Datei werden folgende Anforderungen gestellt:

Zugänglichkeit

1. Die Klasse muss zugänglich (`Public`) sein.

   ```
   Public Class Schnellstart_CB
   ```

Vererbung

2. Die Klasse muss von `System.Web.UI.Page` erben.

   ```
   Inherits System.Web.UI.Page
   ```

Deklaration

3. Für die einzelnen Steuerelemente muss es eine Deklaration geben, damit der Code-Behind-Klasse die Namen bekannt sind. Die Steuerelemente müssen als `Public` oder `Protected` deklariert werden und zusätzlich das Schlüsselwort `WithEvents` beinhalten.

   ```
   Protected WithEvents B_OK As System.Web.UI.WebControls.Button
   ```

Nur wenn diese drei Voraussetzungen erfüllt sind, ist die Bindung der Steuerelemente an die Ereignisbehandlungsroutinen in der Code-Behind-Datei möglich.

Ereignisbindung

Handles

Im Code-Behind-Modell gibt es zusätzlich zur Bindung über die »on«-Attribute (vgl. Ausführungen zum Single-File-Modell) die Möglichkeit zur Bindung über das Schlüsselwort `Handles`.

```
Sub xy() Handles OBJEKTNAME.EREIGNISNAME
```

`Handles` legt in der Code-Behind-Datei fest, für welches Steuerelement-Ereignis diese Routine die Ereignisbehandlung übernimmt. Dies hat zwei Vorteile gegenüber der Bindung im HTML-Code durch ein »on«-Attribut:

1. Der Name der Ereignisbehandlungsroutine muss dem Ersteller der ASPX-Datei (z.B. einem Webdesigner) nicht bekannt sein.
2. Es können beliebig viele Ereignisbehandlungsroutinen für jedes Ereignis definiert werden, da `Handles` mehrfach für das gleiche Ereignis vorkommen darf.

Alternative Ereignisbindung

3. Im Code-Behind-Modell ist aber auch noch die Bindung über die »on«-Attribute möglich. Wenn diese Form gewählt wird, ist zu beachten, dass die Ereignisbehandlungsroutine für die abgeleitete Klasse zugänglich sein muss. Erlaubt sind also die Modifier `Public` und `Protected`, nicht aber `Private`.

Variante	ASPX-Datei	Code-Behind-Datei
Mit »Handles«	`<asp:button id="B_OK" runat="server" Text="OK"> </asp:button>`	`Private Sub B_OK_Click (ByVal sender As System.Object, ByVal e As System.EventArgs) Handles B_OK.Click`
Mit »on«-Attribut	`<asp:button id="B_OK" onclick="B_OK_Geklickt" runat="server" Text="OK"> </asp:button>`	`Public Sub B_OK_Geklickt (ByVal sender As System.Object, ByVal e As System.EventArgs)`

Tabelle 4.3: Alternative Ereignisbindung im Code-Behind-Modell

Mehrfachbindung

Wenn für ein Ereignis sowohl in der Code-Behind-Datei als auch in der ASPX-Seite eine Ereignisbehandlung definiert ist, werden beide ausgeführt. Im .NET Framework kann eine beliebige Anzahl von Ereignisbehandlungsroutinen an ein Ereignis gebunden werden.

4.5 Seitenübergänge

Notwendigkeit für Seitenübergänge

Theoretisch ist es möglich, eine komplette serverseitige Webanwendung in nur einer Datei zu erzeugen, wobei der Programmcode immer wieder andere HTML-Dokumente generiert. Praktisch jedoch würde diese eine Datei extrem lang und aufgrund der notwendigen Fallunterscheidungen unübersichtlich werden. Man teilt daher die Anwendungsdateien besser auf mehrere Dateien auf. Dafür benötigt man dann aber die Möglichkeit, aus einer Datei eine andere aufzurufen und dabei Daten zu übergeben.

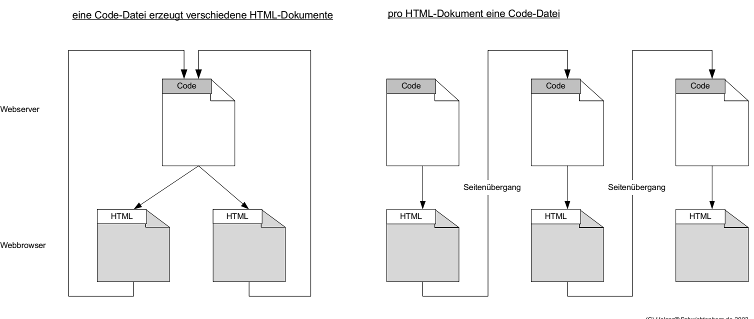

Abbildung 4.11: Alternative Programmierweisen, die aber auch miteinander kombiniert werden können

Grundsätzlich kann man einen Seitenübergang danach unterteilen, ob er vom Client oder vom Server initiiert wird.

Clientseitiger Seitenübergang

1. Ein **clientseitiger Seitenübergang** basiert auf einer Aktion des Anwenders (also einem Klick auf einen Link oder Button) oder auf einer im HTML-Code definierten Weiterleitung per Meta-Tags. In beiden Fällen zeigt der Browser eine Seite an und fragt danach nach einer neuen Seite.

Serverseitiger Seitenübergang

2. Bei einem **serverseitigen Seitenübergang** initiiert der Code auf dem Server, dass nun eine andere als die vom Browser angefragte Seite angezeigt werden muss. Eine Webserver-Middleware hat dazu entweder die Möglichkeit, intern die Kontrolle an eine andere Seite im gleichen Web zu übergeben oder aber den Browser durch einen speziellen HTTP-Status-Code (*302 Object Moved*) anzuweisen, doch »lieber« eine andere Seite anzufragen.

Seitenübergänge in klassischen ASP-Seiten

Im klassischen ASP war es üblich, für die Auswertung eines Formulars zu einer neuen Seite überzugehen. Das Formular war eine reine HTML- oder eine ASP-Seite. Die Auswertungsseite musste eine ASP-Seite sein.

Clientseitiger Seitenübergang

Wenn das Formular selbst eine ASP-Seite war, war es auch möglich, dass es sich selbst aufruft. Dann musste zu Beginn der ASP-Seite geprüft werden, ob die Datei nun zwecks Darstellung des (leeren) Formulars extern aufgerufen wurde oder zwecks Prüfung bzw. Verarbeitung der eingegebenen Werte von sich selbst aufgerufen wurde. Der Übergang zur nächsten Seite erfolgte nach Abschluss der Verarbeitung der Formulardaten.

Serverseitiger Seitenübergang

Das klassische ASP bietet daneben auch zwei Möglichkeiten zum serverseitigen Seitenübergang:

1. `Response.Redirect()` leitet den Browser mit dem HTTP-Rückgabewert *302* zu einer anderen Seite. Wenn Sie Werte an die neue Seite übergeben wollen, müssen Sie einen entsprechenden Querystring zusammenbauen (`Response.Redirect("seitenname.aspx?attribut=wert&...")`) oder aber das State Management nutzen (siehe Kapitel 8).
2. `Server.Transfer()` führt einen reinen serverseitigen Seitenübergang durch, indem die Programmausführung in einer anderen Datei fortgesetzt wird. Werte können hier allerdings nicht per Querystring, sondern nur durch die Möglichkeiten des State Managements übergeben werden.

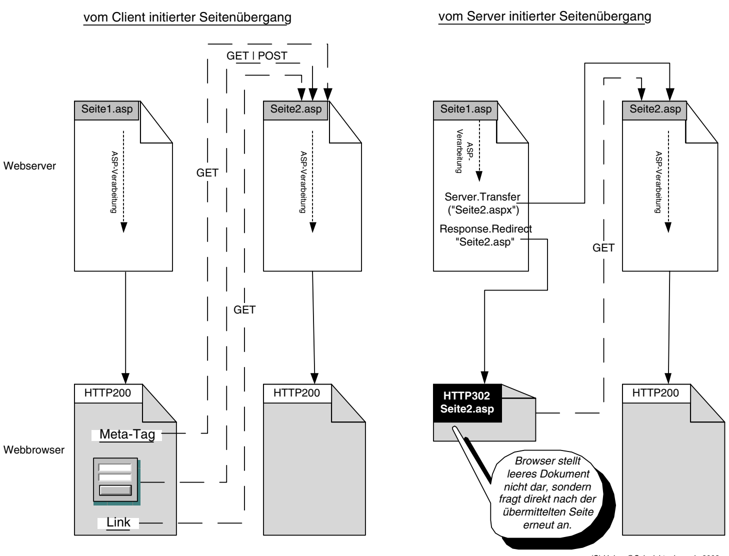

Abbildung 4.12: Seitenübergänge im klassischen ASP

Seitenübergänge in Webforms

Webcontrols laufen serverseitig, was auch zu einer Änderung im Ablaufmodell, also beim Übergang von einer ASPX-Seite (Webform) zur nächsten führt.

Weniger Seitenübergänge

ASP.NET geht in seinem Anwendungsmodell etwas weg von Seitenübergängen und hin zu einem Modell, in dem ASPX-Seiten ihre Angelegenheiten primär selbst regeln. Ein Seitenübergang von einer zur nächsten Webseite erfolgt in ASP.NET so, dass zunächst die aktuelle Seite erneut aufgerufen wird, damit die dort hinterlegten Ereignisbehandlungsroutinen ausgeführt werden können. Erst nachdem diese abgearbeitet wurden, erfolgt (wahlweise serverseitig oder durch einen clientseitigen Redirect) der Aufruf der nächsten Seite. Dies nennt man *Postback-Architektur*.

Ein eventuell vorhandenes Action-Attribut im `<form>`-Tag wird ignoriert, wenn das `<form>`-Tag über das Attribut `runat="server"` verfügt. Das ist auch notwendig, da sonst die Ereignisbehandlungsroutinen gar nicht ausgeführt werden könnten.

Natürlich ist es dennoch notwendig, Konstrukte für den Seitenübergang nach einem Postback zu haben, da man sonst immer nur auf einer Seite verbleiben oder aber dem Benutzer durch einen Hyperlink clientseitig die Möglichkeit zur Navigation geben müsste.

ASP.NET bietet die gleichen zwei Methoden zur durch den Servercode initiierten Client-Navigation (`Response.Redirect()` und `Server.Transfer()`), die es auch schon im klassischen ASP gab. Allerdings ist die Implementierung anders.

Serverseitiger Übergang mit Redirect()

HTTP-Code 302

`Redirect()` leitet genau wie im klassischen ASP den Browser mit dem HTTP-Rückgabewert *302* zu einer anderen Seite. Anders als beim klassischen ASP ist es nicht schlimm, wenn Sie vor einem `Redirect()` schon Ausgaben gemacht haben. ASP.NET kann damit umgehen.

Serverseitiger Übergang mit Transfer()

Von einem `Server.Transfer()` bekommt der Browser dagegen überhaupt nichts mit: Die serverseitige Ausführung wird einfach mit dem Code einer anderen Datei fortgesetzt. Für den Browser ändert sich nicht einmal der URL, denn er bekommt den Inhalt einer anderen Seite unter dem ursprünglich angefragten URL. Für die Übergabe von Variablen zwischen den beiden Seiten bot das klassische `Server.Transfer()` nur die Möglichkeit, Werte in den `Application`- und `Session`-Collections zu speichern; das Anhängen eines Querystrings an den URL war beim Einsatz von `Transfer()` verboten. ASP.NET bietet nun zwei Optionen:

Mit Querystring

1. Anhängen eines Querystrings: Damit ist `Server.Transfer()` aus der Sicht des Entwicklers fast gleich mit einem `Response.Redirect()`. Es bleibt aber der Vorteil, dass kein Roundtrip notwendig ist und der Browser von der Umleitung nichts mitbekommt, weil der URL sich nicht ändert.

Direkter Klassenzugriff

2. Die Folgeseite kann auch direkt auf alle Intrinsic Objects der aufrufenden Seite und auf alle öffentlichen Variablen der zugehörigen Code-Behind-Klasse zugreifen. Die Folgeseite erhält einen Zeiger auf das aufrufende Objekt über `context.Handler`.

> Dass sich der URL im Browser nicht ändert, hat aus der Sicht des Anwenders auch einen Nachteil: Man kann kein Lesezeichen auf die Folgeseite setzen, weil sie aus der Sicht des Browsers den gleichen URL wie die erste Seite hat.

Clientseitiger Übergang

Es gibt auch in ASP.NET ein `HyperLink`-Control, das einen clientseitigen Seitenübergang vollzieht, ohne vorher einen Postback vorzunehmen. Zu beachten ist aber, dass nach einem Klick des Benutzers auf einen solchen Link alle Änderungen an den Formularfeldern verworfen werden.

Außerdem kann ein clientseitiger Übergang natürlich auf Wunsch alternativ durch normale HTML-Tags (<a href="..."> oder <input type="submit">) erreicht werden.

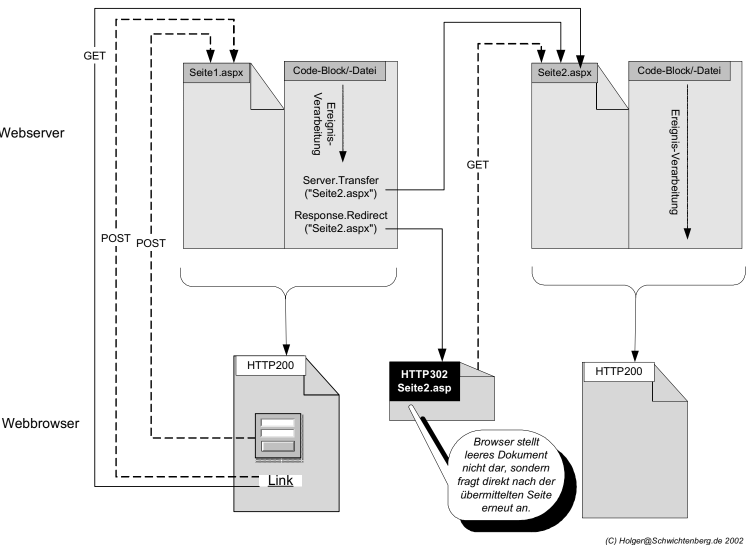

Abbildung 4.13: Seitenübergänge in Webforms

Beispiel

Das folgende Beispiel stellt alle Alternativen zum Seitenübergang in ASP.NET-Webforms gegenüber:

Vier Alternativen

1. Die erste Spalte realisiert den Seitenübergang mit `Transfer()` und direktem Zugriff auf die aufrufende Klasse.
2. Die zweite Spalte verwendet `Transfer()` mit einem Querystring.
3. In der dritten Spalte wird `Redirect()` mit Querystring verwendet.
4. Die vierte Spalte ist ein clientseitiger Übergang, bei dem das `Hyperlink`-Control statt eines `LinkButton` zum Einsatz kommt. Dies ist der einzige Fall, bei dem der Wert des Eingabefeldes nicht an die Folgeseite übermittelt wird.

Mit Ausnahme der letzten Spalte gibt es in jeder Spalte zwei alternative Navigationsmöglichkeiten, einmal per Link und zum anderen per Button. Sie werden aber sehen, dass es dort keinen Unterschied gibt. Den gäbe es lediglich beim clientseitigen Übergang per »Submit«-Button. Das ist jedoch in dieses Beispiel nicht integrierbar, da nur ein <form>-Tag pro ASPX-Seite erlaubt ist.

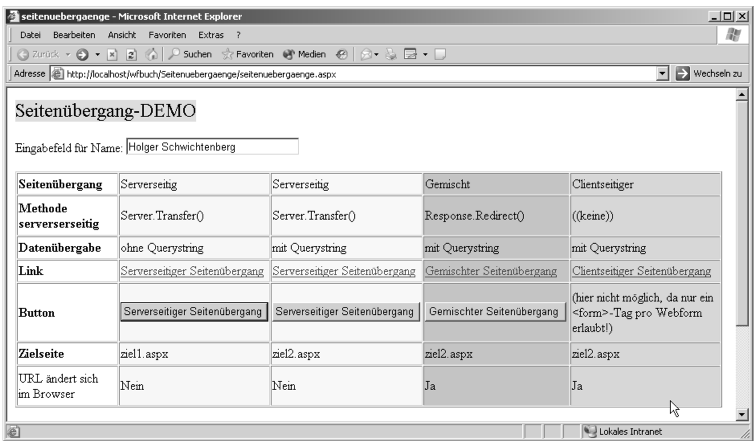

Abbildung 4.14: Demo-Anwendung zum Vergleich der verschiedenen Seitenübergänge [seitenuebergaenge.aspx]

Postback per JavaScript

Die Links in den ersten drei Spalten sind mit der JavaScript-Funktion für den manuellen Postback verknüpft, da ein Link normalerweise keine Formulardaten übermittelt. Der vierte Link ist ein normaler HTML-Hyperlink. Wenn Sie darauf klicken, werden Sie bemerken, dass Ihre Eingabe nicht an *ziel2.aspx* weitergegeben wurde.

Implementierung der aufrufenden Datei

Tags für Webcontrols

Das folgende Listing zeigt nur die Tags für die Webcontrols, da die der Formatierung dienenden HTML-Codes für das Verständnis hier nicht relevant sind.

```
<asp:LinkButton id="LinkButton1" runat="server">
Serverseitiger Seitenübergang</asp:LinkButton>

<asp:LinkButton id="LinkButton2" runat="server">
Serverseitiger Seitenübergang</asp:LinkButton>

<asp:LinkButton id="LinkButton3" runat="server">
Gemischter Seitenübergang</asp:LinkButton>

<asp:HyperLink id="HyperLink" runat="server" NavigateUrl="ziel.aspx">
Clientseitiger Seitenübergang
</asp:HyperLink>

<asp:Button id="Button1" runat="server" Text="Serverseitiger
Seitenübergang" Width="184px"></asp:Button>

<asp:Button id="Button2" runat="server" Text="Serverseitiger
Seitenübergang " Width="184px"></asp:Button>
```

```
<asp:Button id="Button3" runat="server" Text="Gemischter
Seitenübergang " Width="176px"></asp:Button>
```

Listing 4.8: Ausschnitt aus Seitenuebergang.aspx

Code-Behind-Datei

Ereignisbehandlung in aufrufender Seite

Das nächste Listing zeigt die in der zugehörigen Code-Behind-Datei hinterlegten Ereignisbehandlungsroutinen für die Webcontrols in den Spalten 1 bis 3. Dabei wurden jeweils der Button und der Linkbutton an die gleiche Ereignisbehandlungsroutine gebunden. Am Code erkennt man, dass die Spalte 1 auf die Seite *ziel1.aspx* weiterleitet, während die anderen beiden Varianten auf *ziel2.aspx* weiterleiten. Dies ist sinnvoll, da die Verarbeitung im ersten Fall eine andere ist. Wenn es nicht darum geht, den Code möglichst übersichtlich zu halten, könnte man unter Verwendung entsprechender Fallunterscheidungen beide Fälle natürlich in die gleiche Seite nehmen.

```
' --- Spalte 1
Public eingegebenerName As String
  Private Sub LinkButton1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles LinkButton1.Click, Button1.Click
    eingegebenerName = Eingabe.Text
    Server.Transfer("ziel1.aspx")
End Sub

' --- Spalte 2
Private Sub LinkButton2_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles LinkButton2.Click, Button2.Click
    Server.Transfer("ziel2.aspx?eingabe=" & Eingabe.Text)
End Sub

' --- Spalte 3
Private Sub LinkButton3_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles LinkButton3.Click, Button3.Click
    Response.Redirect("ziel2.aspx?eingabe=" & Eingabe.Text)
End Sub
```

Listing 4.9: Ereignisbehandlungsroutinen aus Seitenuebergang.aspx.vb

Implementierung der Folgeseite

Auslesen des Querystrings

Die Implementierung des Zugriffs auf die übergebenen Werte in der aufgerufenen Datei erfolgt im Fall der Übergabe mit Hilfe des Querystrings (Spalte 2 und 3) ganz einfach über das `Request`-Objekt.

```
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    ausgabe.Text = Request("eingabe")
    If ausgabe.Text = "" Then ausgabe.Text = "((nichts))"
End Sub
```

Listing 4.10: ziel2.aspx.vb

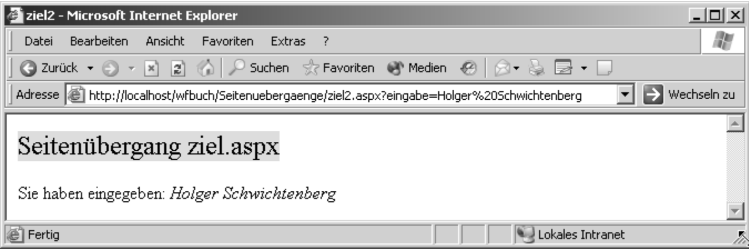

Abbildung 4.15: Der Browser wurde durch das Redirect() auf eine andere Seite umgelenkt [ziel2.aspx].

Direktzugriff

Das nächste Listing zeigt nun, wie *ziel1.aspx* auf das Objekt zugreift, das die Seite mit `Transfer()` aufgerufen hat. `context.Handler` wird auf die Klasse des aufrufenden Objekts (`Seitenuebergaenge`) gecastet, damit frühe Bindung verwendet werden kann.

```
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    ' --- direkter Zugriff auf die aufrufende Klasse
    Dim vorherige As seitenuebergaenge
    vorherige = CType(context.Handler, Seitenuebergaenge)
    ausgabe.Text = vorherige.eingegebenerName
    If ausgabe.Text = "" Then ausgabe.Text = "((nichts))"
End Sub
```

Listing 4.11: ziel1.aspx.vb

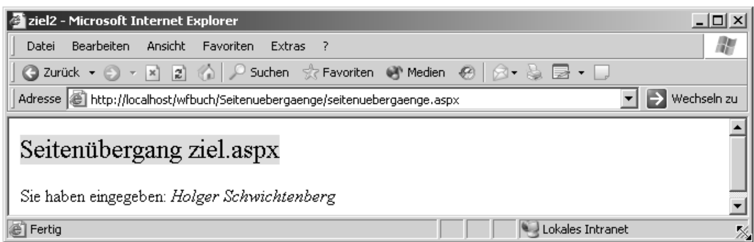

Abbildung 4.16: Beim Einsatz von Transfer() verändert sich der Browser-URL nicht und man sieht keinen Querystring [ziel1.aspx].

Direktzugriff im Single-File-Modell

Direktzugriff im Single-File-Modell

Einen besonderen Trick muss man anwenden, wenn man aus dem Single-File-Modell auf das aufrufende Objekt zugreifen will, da es ja in diesem Fall keine gemeinsame Assembly aller Dateien gibt:

1. Die aufgerufene ASPX-Datei muss die aufrufende ASPX-Datei referenzieren.

```
<%@ Reference Page="aufrufer.aspx"%>
```

2. Man muss als Klassennamen den in der aufrufenden Seite vergebenen Klassennamen verwenden.

```
Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
 Dim vorherige As asp.aufrufer_asp
  vorherige = CType(context.Handler, asp.aufrufer_asp)
  Response.write("Zugriff aus der ASPX-Seite:" &
vorherige.eingegebenerName)
end sub
```

Es bietet sich daher an, in der aufrufenden Seite den Klassennamen manuell zu vergeben.

```
<%@ Page Language="vb" ClassName="xyz" ...%>
```