Building ASP.NET Server Controls

DALE MICHALK AND ROB CAMERON

Apress™

Building ASP.NET Server Controls
Copyright © 2004 by Dale Michalk and Rob Cameron

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-140-2

Printed and bound in the United States of America 10987654321

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: Larry Wall

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Copy Editor: Nicole LeClerc

Production Manager: Kari Brooks

Proofreader: Linda Seifert

Compositor: Diana Van Winkle, Van Winkle Design Group

Indexer: Kevin Broccoli

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 5

Event-Based Programming

IN THIS CHAPTER, we explore the intricacies of working with server control events. The first part of this chapter is a general discussion of the .NET event architecture. We discuss how to add events to a control, bringing back our favorite `Textbox` control as part of the demonstration. Then we illustrate how to define custom events and add them to yet another version of our famous `Textbox`. We also examine `System.Web.UI.Control`'s support for maintaining events. Next, we show how to initiate and capture a postback using a `Button` control that we create named `SuperButton`. This section examines `Command` events and event bubbling with an example composite control to demonstrate these concepts. In the final portion of the chapter, we bring it all together with a discussion of the page life cycle, focusing on events. The next section provides an overview of events and ASP.NET controls.

Events and ASP.NET Controls

The event-based development paradigm is a well-traveled path on the Windows platform with Visual Basic 6.0 and Visual C++ Microsoft Foundation Classes (MFC) development tools. In this model, developers need not be concerned with the details of how to gather input from hardware or render output to the video card; instead, they can focus on business logic coded in event handlers attached to UI widgets that receive events from the operating system. In ASP.NET, this development model is brought to the Web in much the same way through server controls.

The key technology that sets ASP.NET apart from previous web development paradigms is the use of server-side controls as first-class objects in a similar fashion to Visual Basic or MFC. Server controls provide a rich, object-oriented method of building web content in an environment that is normally spartan in its feature set and procedural in its execution model. A critical aspect of working with objects such as ASP.NET server controls is event-based programming, which we cover in this chapter.

The Need for Events in ASP.NET

In any object-oriented development framework, events are a necessary means of decoupling reusable functionality from the specifics of any given application. This is true in ASP.NET as well. Events allow the encapsulated functionality of a server control, such as a `Button`, to be hooked into the logic of an application without requiring any changes, such as recompilation, to the UI object.

Events simplify the work of the programmer by providing a consistent protocol for development. Client applications can register their interest in a control via an event and be notified later by the control when some activity has taken place in the same way

regardless of the control. The only thing that changes from control to control is the number or type of events that are available as well as possibly the arguments that a particular event makes available in its method signature. Figure 5-1 presents a comparison between traditional programming and event-based programming.

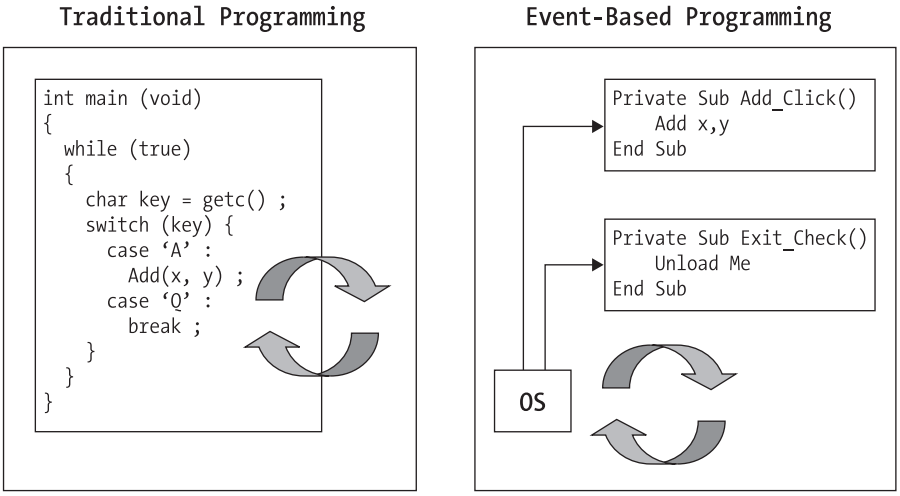

Figure 5-1. Traditional programming versus event-based programming

ASP.NET turns on its head the traditional assumption that UI controls are only appropriate for thick-client applications. Through clever use of client-side state and the HTTP POST protocol, ASP.NET server controls appear as if they maintain memory on the client and react to user interaction by raising events. Server controls do this without having to resort to a bunch of client-side tricks such as applets or ActiveX controls. Even browsing devices that don't support JavaScript on the client can raise events through HTML form actions.

Figure 5-2 illustrates how a control can raise events and make it look like ASP.NET has turned the browser into an interactive thick-client application. The `TextBox` exposes the `TextChanged` event, while the `Button` notifies interested clients through a `Click` event. All event-handling code for the `TextChanged` and `Click` events is located on the server where the ASP.NET processing occurs.

For the event code to react to changes the user makes with the `TextBox` on the Web Form in the browser, the control must shift execution from the browser back to the web server. The `Button` control is responsible for handling this by generating a form postback when it is clicked.

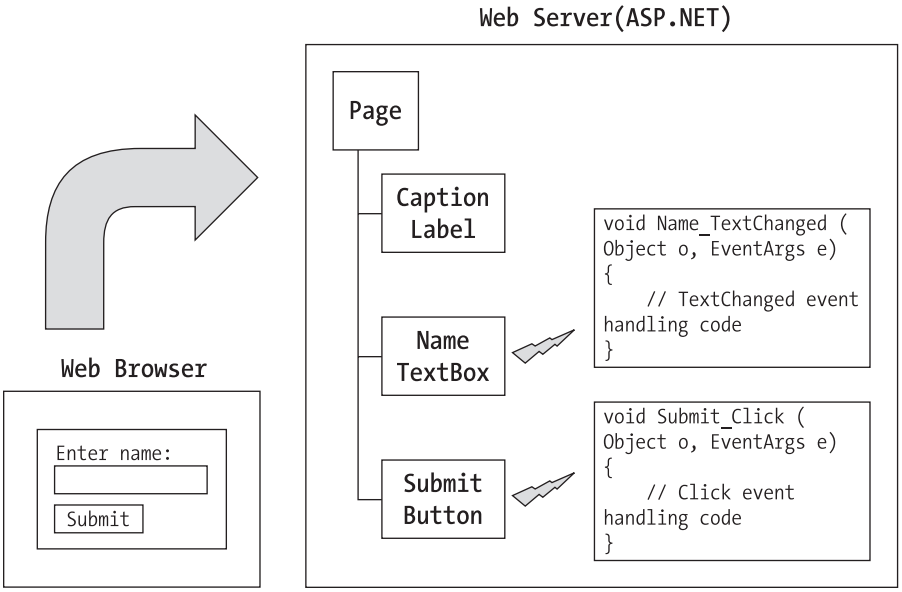

Figure 5-2. Server-side control events in ASP.NET

`Buttons` automatically generate a form postback, but other server controls can also generate a postback using JavaScript. Changing the `AutoPostBack` property for a control from the default value of `false` to `true` will cause the control to emit the appropriate JavaScript, taking advantage of client-side events to cause postback.

When a user clicks a `Button` in a Web Form and the browser performs an HTTP POST back to the web server, ASP.NET builds the page control tree on the server to dynamically handle the postback request. As discussed in Chapter 4, ASP.NET gives each control in the control tree that has ViewState enabled a chance to examine posted data through its `LoadPostData` method. In this method, the control can examine the user input on the server via the posted data and compare it to what was previously stored in ViewState. If the data has indeed changed, and if the control has an event that can be fired in response to the data change, the control should return true to ASP.NET in `LoadPostData`. Later on in the page life cycle, ASP.NET will call the `RaisePostDataChangedEvent` member for each server control that returned true in `LoadPostData` so that the control can in turn raise the appropriate event and execute any business logic implemented via an event handler written by the developer. This is the ASP.NET page life cycle that repeats during each postback for state (data) changes and event firing. Before we dive into writing events for ASP.NET, we first provide a high-level overview of events in the .NET Framework in the next section.

The .NET Framework Event Model

Events are generally used in UI development to notify the appropriate object that the user has made a selection, but events can be used for any asynchronous communication need. Whether you're developing desktop Windows applications using Windows Forms or web applications using ASP.NET, classes as objects need a mechanism to communicate with each other. The .NET Framework provides a first-class event model based on delegates, which we discuss in this section.

Delegates

Delegates are similar to interfaces—they specify a contract between the publisher and the subscriber. Although an interface is generally used to specify a set of member functions, a delegate specifies the signature of a single function. To create an instance of a delegate, you must create a function that matches the delegate signature in terms of parameters and data types.

Delegates are often described as safe function pointers; however, unlike function pointers, delegates call more than one function at a time and can represent both static and instance methods. Also unlike function pointers, delegates provide a type-safe callback implementation and can be secured through code access permissions as part of the .NET Framework security model.

Delegates have two parts in the relationship: the delegate declaration and the delegate instance or static method. The *delegate declaration* defines the function signature as a reference type. Here is how a delegate is declared:

```
public delegate int PrintStatusNotify (object printer, object document) ;
```

Delegates can be declared either outside a class definition or as part of a class through the use of the `delegate` keyword. The .NET Framework `Delegate` class and the .NET Framework `MulticastDelegate` class serve as the base classes for delegates, but neither of these classes is creatable by developers; instead, developers use the `delegate` keyword. As background, the `MulticastDelegate` base class maintains a linked list of delegates that are invoked in order of declaration when the delegate is fired, as you will see in our example in the next section.

One question that arises with delegates is what happens if an invoked method throws an exception. Does the delegate continue processing the methods in the invocation list? Actually, if an exception is thrown, the delegate stops processing methods in the invocation list. It does not matter whether or not an exception handler is present. This makes sense because odds are that if an invoked method throws an exception, methods that follow may throw an exception as well, but it is something to keep in mind.

Working with Delegates

In this section we create a console-based application to demonstrate how delegates work. In our example, we declare a very simple delegate that takes one parameter:

```
delegate void SimpleMulticastDelegate(int i);
```

We next declare a class that contains two class instance methods and one static method. These methods match the signature of the previous delegate declaration:

```
public class DelegateImplementorClass
{
    public void ClassMethod(int i)
    {
        Console.WriteLine("You passed in " + i.ToString() +"
                          to the class method");
    }

    static public void StaticClassMethod(int j)
    {
        Console.WriteLine("You passed in "+ j.ToString() +"
         to the static class method");
    }

    public void YetAnotherClassMethod(int k)
    {
        Console.WriteLine("You passed in " + k.ToString() +"
        to yet another class method");
    }
}
```

In method `Main`, the entry point of any console application in .NET, we put the delegate to work. Here we declare an instance of `DelegateImplementorClass`, as we will add instance methods from this class as subscribers to our delegate:

```
DelegateImplementorClass ImpClass = new DelegateImplementorClass();
```

We next declare an instance of our delegate, adding an instance method to the delegate invocation list that will be called when the delegate instance executes:

```
SimpleMulticastDelegate d = new SimpleMulticastDelegate(ImpClass.ClassMethod);
```

Firing the delegate is simply a matter of calling the delegate instance function:

```
d(5);
```

The rest of method `Main` adds additional methods to the delegate's invocation list. Listing 5-1 is the full code listing. Figure 5-3 shows the output. Notice how each subsequent call to the delegate reflects this in the output. Each time the delegate fires, it passes the parameter value to each subscriber in its invocation list, taking advantage of multicasting behavior.

Listing 5-1. Delegates in Action

```
using System;

namespace ControlsBookWeb.Ch05
{
   delegate void SimpleMulticastDelegate(int i);

   public class DelegateImplementorClass
   {
      public void ClassMethod(int i)
      {
         Console.WriteLine("You passed in " + i.ToString() +"
         to the class method");
      }

      static public void StaticClassMethod(int j)
      {
         Console.WriteLine("You passed in "+ j.ToString() +"
          to the static class method");
      }

      public void YetAnotherClassMethod(int k)
      {
          Console.WriteLine("You passed in " + k.ToString() +"
           to yet another class method");
      }
   }

   class main
   {
      [STAThread]
      static void Main(string[] args)
      {
         DelegateImplementorClass ImpClass = new DelegateImplementorClass();

         SimpleMulticastDelegate d =
              new SimpleMulticastDelegate(ImpClass.ClassMethod);
         d(5);
         Console.WriteLine("");

         d +=
         new SimpleMulticastDelegate(DelegateImplementorClass.StaticClassMethod);
         d(10);
         Console.WriteLine("");

         d += new SimpleMulticastDelegate(ImpClass.YetAnotherClassMethod);
         d(15);
      }
   }
}
```

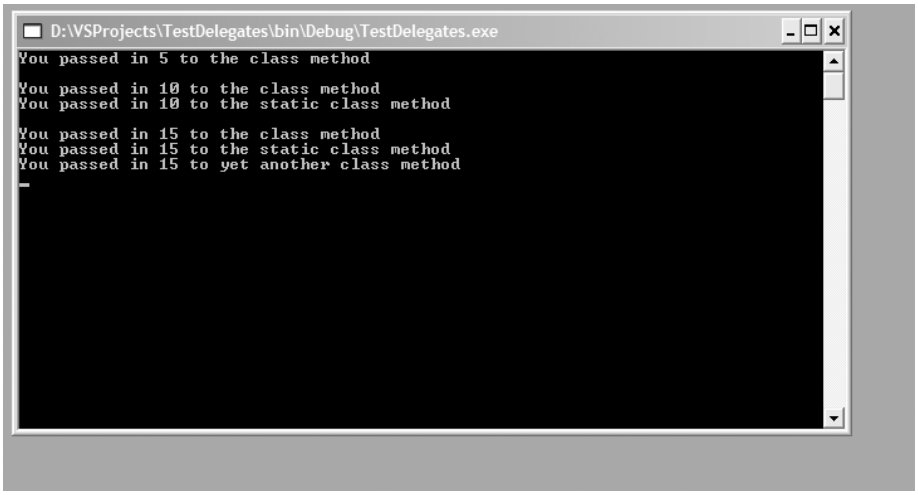

Figure 5-3. Output from our work with delegates

Stepping back for a minute, you can see how delegates quite successfully fulfill the requirements of the publisher/subscriber model. Here we have the member function `Main` using an instance of the delegate to send messages to subscribing methods in the `DelegateImplementorClass` class. As long as the subscribing methods match the delegate signature, the delegate is happy to add those methods to its invocation list, and it promptly processes this list each time it is invoked with a call to `d()`.

If you step through this code with the debugger, you will notice that methods on the delegate's invocation list are synchronously called in the order that they are added to the invocation list. The syntax for adding a delegate to the invocation list may seem strange at first, because what we are really adding is something more akin to function pointers than, say, an integer. The magic behind this is the keyword `delegate` and the .NET infrastructure provided by the `System.Delegate` and `System.Delegate.MulticastDelegate` classes. The result is that the language compiler simplifies things by providing a keyword that developers use to plug into the delegate infrastructure.

Events

As you may have guessed by now, delegates are the heart and soul of event handling in .NET. They provide the underlying infrastructure for asynchronous callbacks and UI events in web applications under ASP.NET. In addition to the `delegate` keyword, there is also the `event` keyword in C#. The `event` keyword lets you specify a delegate that will fire upon the occurrence of some event in your code. The delegate associated with an event can have one or more client methods in its invocation list that will be called when the object indicates that an event has occurred, as is the case with a `MulticastDelegate`.

We can declare an event using the `event` keyword followed by a delegate type and the name of the event. The following event declaration creates a `Click` event with public accessibility that would be right at home on a `Button` control:

```
public event EventHandler Click;
```

The name of the event should be a verb signifying that some action has taken place. `Init`, `Click`, `Load`, `Unload`, and `TextChanged` are all good examples of such verbs used in the ASP.NET framework.

The event declaration causes the C# compiler to emit code that adds a private field to the class named `Click`, along with add and remove methods for working with the delegates passed in from clients. The nice thing about the event declaration and the code it generates is that it happens under the covers without your having to worry about it. Later on in this chapter, we discuss how to optimize event registration with respect to storage for controls that publish a large number of events, but only a small fraction of them are likely to be subscribed to for a given control instance.

System.EventHandler Delegate

The common denominator of the event declarations with .NET controls is the delegate class `System.EventHandler`. All the built-in controls in ASP.NET use its signature or some derivative of it to notify their clients when events occur. We recommend that you leverage this infrastructure because it reduces the amount of custom event development required. In addition, the signature of `EventHandler` permits server controls in the .NET Framework and their clients to interoperate:

```
delegate void EventHandler(Object o, EventArgs e);
```

The first parameter to `EventHandler` is an object reference to the control that raised the event. The second parameter to the delegate, `EventArgs`, contains data pertinent to the event. The base `EventArgs` class doesn't actually hold any data; it's more of an extensibility point for custom events to override. The `EventArgs` class does have a read-only static field named `Empty` that returns an instance of the class that's syntactically convenient to use when raising an event that doesn't require any special arguments or customization.

Invoking an Event in a Control

After you add an event to a control, you need to raise the event in some manner. Instead of calling the event directly, a good design pattern followed by all the prebuilt server controls in ASP.NET is to add a `virtual protected` method that invokes the event with a prefix of `On` attached to the name of the method. This provides an additional level of abstraction that allows controls that derive from a base control to easily override the event-raising mechanism to run additional business logic or suppress event invocation altogether. The following code shows an `OnClick` protected method used to provide access to the `Click` event of class:

```
protected virtual void OnClick(EventArgs e)
{
   if (Click != null)
      Click(this, e);
}
```

The first thing the protected method does is check to see if any client methods have registered themselves with the `Click` event instance. The event field will have a `null` value if no clients have registered a method onto the delegate's invocation list. If clients have subscribed to the `Click` event with a method having a matching signature, the event field will contain an object reference to a delegate that maintains the invocation list of all registered delegates. The `OnClick` routine next invokes the event using the function call syntax along with the name of the event. The parameters passed in are a reference to the control raising the event and the event arguments passed into the routine.

Adding an Event to the Textbox Control

The `Textbox` control that we started in Chapter 4 had the beginnings of a nice clone of the ASP.NET `System.Web.UI.WebControls.TextBox` control. It saves its values to ViewState, emits the correct HTML to create a text box in the browser, and handles postback data correctly. The control is well on its way to becoming a respectable member of the family.

We next enhance our `Textbox` control by adding the capability to raise an event when the `Text` property of the control has changed, as detected by comparing the value currently stored in ViewState with postback data.

Enhancing the Textbox Control with a TextChanged Event

The next step in our `Textbox` journey is to add a `TextChanged` event to help bring its functionality more in line with that of the built-in ASP.NET text controls. This necessitates adding an event declaration and enhancing the implementation of the `IPostBackDataHandler` interface in our control. The most important upgrade is the addition of the `TextChanged` event field and a protected `OnTextChanged` method to invoke it:

```
protected virtual void OnTextChanged(EventArgs e)
{
   if (TextChanged != null)
      TextChanged(this, e);
}
public event EventHandler TextChanged;
```

The second upgrade is the logic enhancement to the `LoadPostData` and `RaisePostDataChanged` methods. In `LoadPostData`, the ViewState value of the `Text` property is checked against the incoming value from postback for any differences.

If there is a difference, the Text property is changed to the new value in ViewState and true is returned from the routine. This guarantees that the event is raised when RaisePostDataChangedEvent is called by ASP.NET further on in the page life cycle.

```
public bool LoadPostData(string postDataKey, NameValueCollection postCollection)
{
   string postedValue = postCollection[postDataKey];
   if (!Text.Equals(postedValue))
   {
      Text = postedValue;
      return true;
   }
   else
      return false;
}
```

The upgrade to the RaisePostDataChangedEvent method is the addition of a single line. Instead of being blank, it calls on our newly created OnTextChanged method to invoke the TextChanged event. We use the static field Empty of the EventArgs class to create an instance of EventArgs for us, as we don't need to customize EventArgs in this case:

```
public void RaisePostDataChangedEvent()
{
   OnTextChanged(EventArgs.Empty);
}
```

The code in Listing 5-2 is full text of the control after the modifications required to add the TextChanged event.

Listing 5-2. Event-Based Programming

```
using System;
using System.Web;
using System.Web.UI;
using System.Collections.Specialized;
using System.ComponentModel;

namespace ControlsBookLib.Ch05
{
   [ToolboxData("<{0}:Textbox runat=server></{0}:Textbox>"),
   DefaultProperty("Text")]
   public class Textbox : Control, IPostBackDataHandler
   {
      public virtual string Text
      {
         get
         {
            object text = ViewState["Text"];
```

```
            if (text == null)
               return string.Empty;
            else
               return (string) text;
         }
         set
         {
            ViewState["Text"] = value;
         }
      }

      public bool LoadPostData(string postDataKey,
         NameValueCollection postCollection)
      {
         string postedValue = postCollection[postDataKey];
         if (!Text.Equals(postedValue))
         {
            Text = postedValue;
            return true;
         }
         else
            return false;
      }

      public void RaisePostDataChangedEvent()
      {
         OnTextChanged(EventArgs.Empty);
      }

      protected virtual void OnTextChanged(EventArgs e)
      {
         if (TextChanged != null)
            TextChanged(this, e);
      }

      public event EventHandler TextChanged;

      override protected void Render(HtmlTextWriter writer)
      {
         base.Render(writer);
         Page.VerifyRenderingInServerForm(this);
         // write out the <INPUT type="text"> tag
         writer.Write("<INPUT type=\"text\" name=\"");
         writer.Write(this.UniqueID);
         writer.Write("\" value=\"" + this.Text + "\" />");
      }
   }
}
```

Using the Textbox Control on a Web Form

The Textbox Web Form shown in the Design view in Figure 5-4 hosts the newly minted `Textbox` control with its `TextChanged` event capabilities.

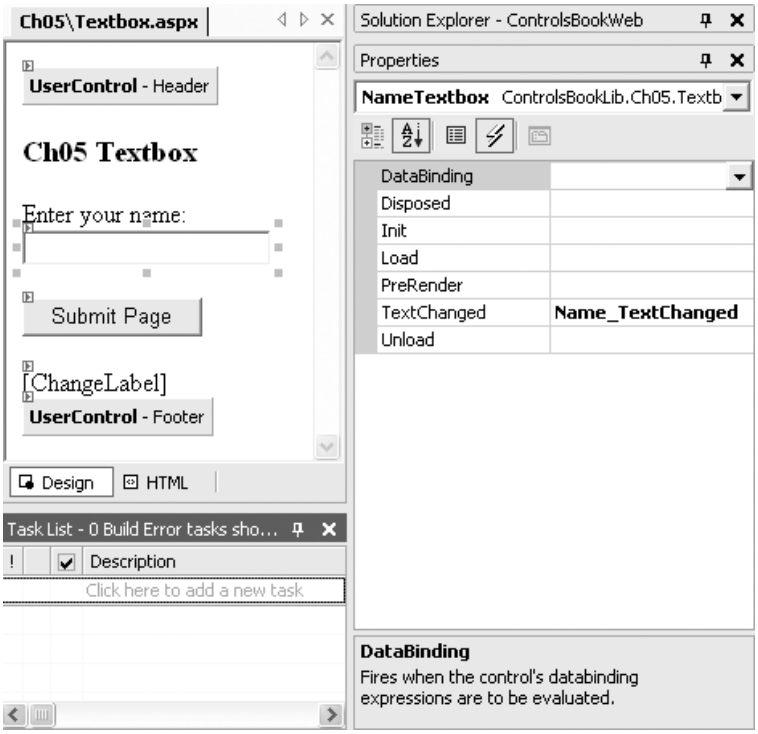

Figure 5-4. Server-side control events in ASP.NET

The Web Form contains an instance of our `Textbox` control named `NameTextbox`, along with a `Label` control named `ChangeLabel` that is used to indicate the raising of `TextChanged` event. The label is programmatically set to a value of "No change!" along with the current time by default during the loading of the Web Form. Raising the `TextChanged` event causes the event-handling code to set the label's value to "Changed." along with the current time. This allows you to recycle the control several times to verify that the event is working properly.

The `TextChanged` event of the `NameTextbox` control is visible when you select the control in the Design view of Visual Studio .NET and look at it in the Properties window, as shown in Figure 5-5. Click the lightning bolt icon to categorize the properties by events and you will see `TextChanged`. We used an event handler called `Name_TextChanged` as a client subscriber to the `TextChanged` event. The full extent of our code work is shown in Listings 5-3 and 5-4.

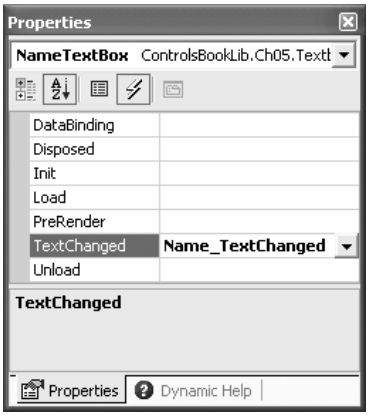

Figure 5-5. The Server-side events tab of the Properties window

Listing 5-3. The Textbox Web Form .aspx Page File

```
<%@ Page language="c#" Codebehind="Textbox.aspx.cs" AutoEventWireup="false"
Inherits="ControlsBookWeb.Ch05.Textbox" %>
<%@ Register TagPrefix="apress" Namespace="ControlsBookLib.Ch05"
Assembly="ControlsBookLib" %>
<%@ Register TagPrefix="apressUC" TagName="ControlsBookHeader"
Src="..\ControlsBookHeader.ascx" %>
<%@ Register TagPrefix="apressUC" TagName="ControlsBookFooter"
Src="..\ControlsBookFooter.ascx" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>Ch05 Textbox</title>
    <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
     content="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>
  <body MS_POSITIONING="FlowLayout">
    <form id="Textbox" method="post" runat="server">
      <apressUC:ControlsBookHeader id="Header" runat="server"
       ChapterNumber="5" ChapterTitle="Event-based Programming" />
      <h3>Ch05 TextBox</h3>
      Enter your name:<br>
      <apress:textbox id="NameTextbox" runat="server"></apress:textbox><br>
      <br>
      <asp:button id="SubmitPageButton" runat="server" Text=
         "Submit Page"></asp:button><br>
      <br>
      <asp:label id="ChangeLabel" runat="server" Text=""></asp:label><br>
      <apressUC:ControlsBookFooter id="Footer" runat="server" />
    </form>
  </body>
</HTML>
```

Listing 5-4. The Textbox Web Form Code-Behind Class File

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ControlsBookWeb.Ch05
{
   public class Textbox : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.Button SubmitPageButton;
      protected ControlsBookLib.Ch05.Textbox NameTextbox;
      protected System.Web.UI.WebControls.Label ChangeLabel;

      private void Page_Load(object sender, System.EventArgs e)
      {
         ChangeLabel.Text = DateTime.Now.ToLongTimeString() + ": No change.";
      }

      #region Web Form Designer generated code
      override protected void OnInit(EventArgs e)
      {
         //
         // CODEGEN: This call is required by the ASP.NET Web Form Designer.
         //
         InitializeComponent();
         base.OnInit(e);
      }

      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.NameTextbox.TextChanged += new
         System.EventHandler(this.Name_TextChanged);
         this.Load += new System.EventHandler(this.Page_Load);

      }
      #endregion

      private void Name_TextChanged(object sender, System.EventArgs e)
      {
         ChangeLabel.Text = DateTime.Now.ToLongTimeString() + ": Changed!";
      }
   }
}
```

The event wiring is conducted inside the `InitializeComponent` routine:

```
private void InitializeComponent()
{
   this.NameTextbox.TextChanged += new System.EventHandler(this.Name_TextChanged);
   this.Load += new System.EventHandler(this.Page_Load);
}
```

The `Name_TextChanged` method is wrapped by a `System.EventHandler` delegate and then passed to the `TextChanged` event of our custom `Textbox` control to add it to its delegate invocation list. Notice also the wiring up of the `Page_Load` method to the `Load` event that the `Page` class will raise during request processing.

The execution of the Web Form during the initial page request results in the UI output of Figure 5-6. The ViewState rendered by the control into this Web Form shows the `Text` property as a blank value. We entered a name into the `Textbox` as well, but we haven't clicked the button to submit the Web Form via postback.

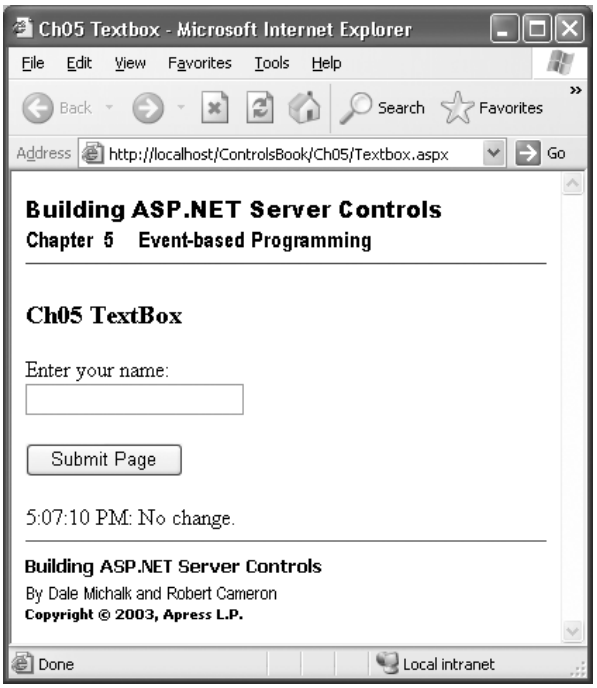

Figure 5-6. Initial rendering of the `Textbox` *control*

Upon clicking the button to execute a postback to the web server, the `Textbox` control will read the blank value from ViewState and find the name value "Dale Michalk" when the ASP.NET invokes `LoadPostData`. Because the posted data is different from the current ViewState value, it calls its internal `OnTextChanged` method to raise events to all

registered delegate subscribers. This results in the `Name_TextChanged` event handler method firing, and the code that changes the label to reflect the new value executes:

```
private void Name_TextChanged(object sender, System.EventArgs e)
{
   ChangeLabel.Text = DateTime.Now.ToLongTimeString() + ": Changed!";
}
```

The result is that `ChangeLabel` displays the text containing the current time and the word "Changed!" as shown in Figure 5-7.

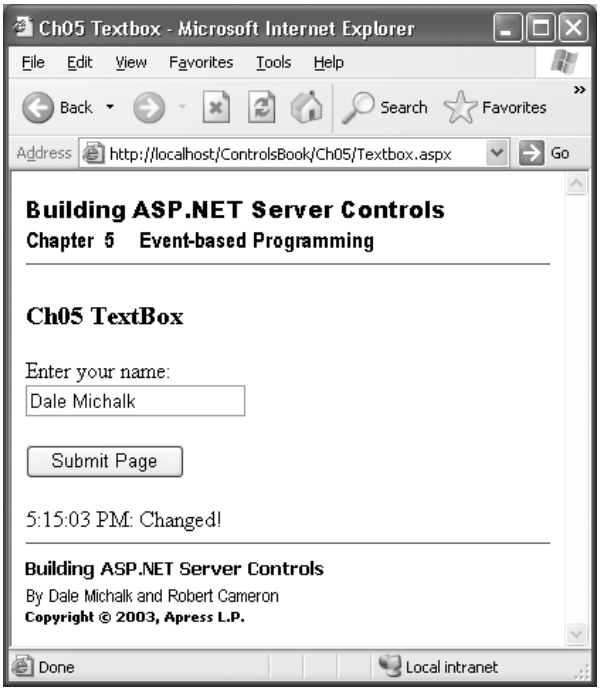

Figure 5-7. The firing of the `TextChanged` *event by the* `Textbox` *control*

The next step in this demonstration is to recycle the page without changing the value in the `Textbox` control by simply clicking the Submit Page button. Because the ViewState and the control's text post data contain the same value of "Dale Michalk," no event is raised. The increment of the timestamp in the label in Figure 5-8 confirms that the page was processed successfully. Our control is able to react appropriately to changes of its `Text` property.

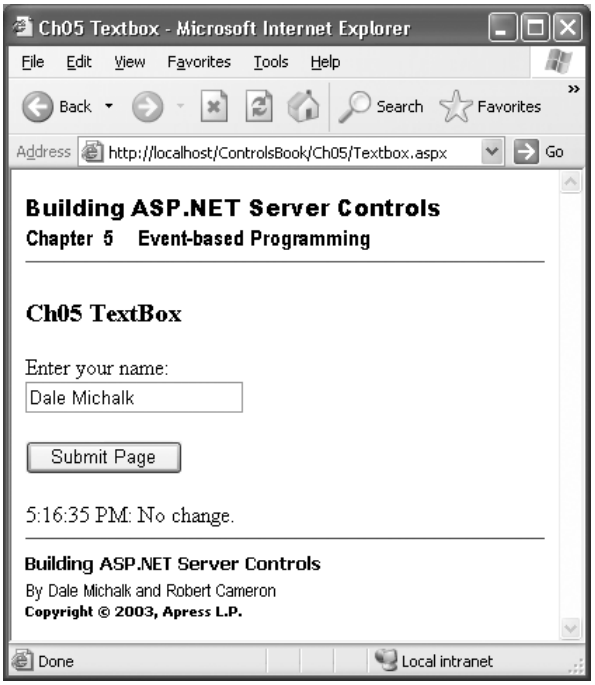

Figure 5-8. The firing of the `TextChanged` *event by the* `Textbox` *control*

Creating a Custom Event

If an event does not provide data but is merely a signal that something has happened, you can take advantage of the `EventHandler` delegate class and its empty `EventArgs` implementation. However, we want to provide additional information in the `TextChanged` event raised by our `Textbox` control. The newly minted event will track both before and after values of the `Text` property between postback submissions. The control loads the `oldValue` from data saved in ViewState; the `newValue` value loads from the data received in the `<INPUT type="text">` HTML element through postback. We now move on to create our custom `EventArgs` class to support our custom event.

Creating a TextChangedEventArgs Class

The first requirement is to create an enhanced `EventArgs`-based class that holds the event data. We create a new class derived from `EventArgs` that exposes two read-only properties to clients, `OldValue` and `NewValue`, as shown in the following code:

```
public class TextChangedEventArgs : EventArgs
{
   private string oldValue;
   private string newValue;
```

```
    public TextChangedEventArgs(string oldValue, string newValue)
    {
        this.oldValue = oldValue;
        this.newValue = newValue;
    }

    public string OldValue
    {
        get
        {
            return oldValue;
        }
    }

    public string NewValue
    {
        get
        {
            return newValue;
        }
    }
}
```

The class created is fairly straightforward. The two properties have only `get` accessors to make them read-only, making the constructor the only way to populate the internal fields with their values.

Creating a TextChangedEventHandler Delegate

Delegate creation is the next step in defining our custom event. There is not an inheritance chain that must be followed with delegates, as all delegate types are created using the keyword `delegate`. Instead, we choose to follow the method signature used by other controls in ASP.NET to build upon a successful design pattern.

The signature of the delegate has two parameters and a `void` return value. The first parameter remains of type `object`, and the second parameter must be of type `EventArgs` or derived from it. Because we already created the `TextChangedEventArgs` class, we use that as our second parameter to take advantage of its `OldValue` and `NewValue` properties.

The name used in the declaration of the following delegate is also important. The pattern for ASP.NET controls is to add the word "EventHandler" to the end of the event of the delegate. In this case, we add "TextChanged" to "EventHandler" to get `TextChangedEventHandler` as our name.

Both the `TextChangedEventArgs` class and the `TextChangedEventHandler` delegate are put into a file named TextChanged.cs that is part of the ControlsBookLib library project for reference by our new control, as shown in Listing 5-5.

Listing 5-5. The `TextChanged` *Class File for the* `TextChangedEventArgs` *Class and* `TextChangedEventHandler` *Delegate Definitions*

```
using System;

namespace ControlsBookLib.Ch05
{
   public delegate void
          TextChangedEventHandler(object o, TextChangedEventArgs tce);

   public class TextChangedEventArgs : EventArgs
   {
      private string oldValue;
      private string newValue;

      public TextChangedEventArgs(string oldValue, string newValue)
      {
         this.oldValue = oldValue;
         this.newValue = newValue;
      }

      public string OldValue
      {
         get
         {
            return oldValue;
         }
      }

      public string NewValue
      {
         get
         {
            return newValue;
         }
      }
   }
}
```

Adding an Event to the CustomEventTextbox Control

To demonstrate the newly minted `TextChangedEventHandler` delegate, we take our `Textbox` control and copy its contents into a class named `CustomEventTextbox`. Another option would be to customize the behavior in an object-oriented manner by overriding the necessary methods in a derived class. However, in this chapter, we choose the route of separate classes so that we can more clearly isolate the two `Textbox` control examples and highlight the different design decisions embodied in them.

Replacing the event declaration is the easiest part. The control starts with an `EventHandler` delegate but is changed to take a `TextChangedEventHandler` delegate:

```
public event TextChangedEventHandler TextChanged;
```

The second change is the replacement of the `OnTextChanged` event invocation method to take `TextChangedEventArgs` as the single parameter to the method, as shown in the following code. This is one of the reasons for having the `On`-prefixed methods in controls as an abstraction layer. It makes it a simpler code change to augment or replace the event mechanism.

```
protected virtual void OnTextChanged(TextChangedEventArgs tce)
{
   if (TextChanged != null)
      TextChanged(this, tce);
}
```

The next step is to add logic to track the before and after values. A private string field named `oldText` is added to the class and is given its value inside `LoadPostData`. This gives us a chance to load `TextChangedEventArgs` properly when we raise the event. Here is a snippet of the code change from `LoadPostData` that does the work:

```
if (!Text.Equals(postedValue))
{
   oldText = Text;
   Text = postedValue;
   return true;
}
```

The last step is to replace all routines that call `OnTextChanged`. We have only one: `RaisePostDataChanged`. It takes the before and after values from the `oldText` field and the `Text` property in `LoadPostData` and creates a new `TextChangedEventArgs` class instance:

```
public void RaisePostDataChangedEvent()
{
   OnTextChanged(new TextChangedEventArgs(oldText, Text));
}
```

Our control is now ready for testing on a Web Form to display its dazzling event capabilities. Listing 5-6 contains the full source code.

Listing 5-6. The `CustomEventTextbox` *Control Class File*

```
using System;
using System.Web;
using System.Web.UI;
using System.Collections.Specialized;
using System.ComponentModel;

namespace ControlsBookLib.Ch05
{
   [ToolboxData("<{0}:CustomEventTextbox runat=server></{0}:CustomEventTextbox>"),
   DefaultProperty("Text")]
   public class CustomEventTextbox : Control, IPostBackDataHandler
   {
      private string oldText;

      public virtual string Text
      {
         get
         {
            object text = ViewState["Text"];
            if (text == null)
               return string.Empty;
            else
               return (string) text;
         }
         set
         {
            ViewState["Text"] = value;
         }
      }

      public bool LoadPostData(string postDataKey,
         NameValueCollection postCollection)
      {
         string postedValue = postCollection[postDataKey];
         if (!Text.Equals(postedValue))
         {
            oldText = Text;
            Text = postedValue;
            return true;
         }
         else
            return false;
      }
```

```
        public void RaisePostDataChangedEvent()
        {
           OnTextChanged(new TextChangedEventArgs(oldText, Text));
        }

        protected void OnTextChanged(TextChangedEventArgs tce)
        {
           if (TextChanged != null)
              TextChanged(this, tce);
        }
        public event TextChangedEventHandler TextChanged;

        override protected void Render(HtmlTextWriter writer)
        {
           base.Render(writer);
           Page.VerifyRenderingInServerForm(this);
           // write out the <INPUT type="text"> tag
           writer.Write("<INPUT type=\"text\" name=\"");
           writer.Write(this.UniqueID);
           writer.Write("\" value=\"" + this.Text + "\" />");
        }
     }
}
```

Using the CustomEventTextbox Control on a Web Form

After building our new control, we are ready to put it to use in the CustomEventTextbox Web Form. This Web Form has the `CustomEventTextbox` control plus a button and two labels named `BeforeLabel` and `AfterLabel` that are used to track the before and after values of the control when the custom `TextChanged` event is raised.

Creating the event mapping in Visual Studio .NET is performed in the same manner as the previous `TextChanged` event in the preceding `Textbox` demonstration. We use the Properties window as shown in Figure 5-9 to wire up the event to the `NameCustom_TextChanged` handling method in the code-behind class.

The Web Form starts out with the labels displaying blank values, as shown in Figure 5-10. We enter Dale's name to cause the next form submit to raise the event. Listings 5-7 and 5-8 contain the source code for the CustomEventTextbox Web Form.

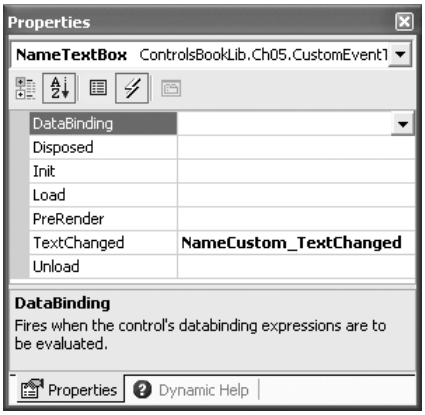

Figure 5-9. The Properties window view of our custom `TextChanged` *event*

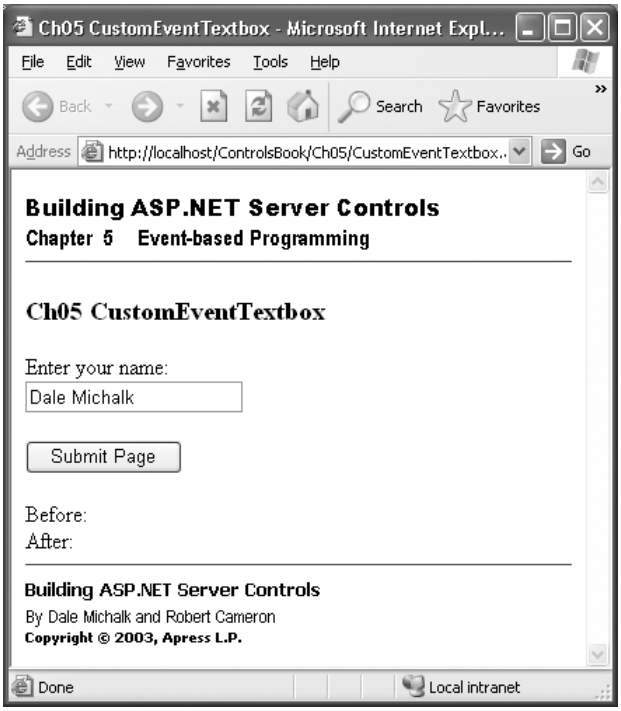

Figure 5-10. Initial page request with the CustomEventTextbox Web Form

Listing 5-7. The CustomEventTextbox Web Form .aspx Page File

```
<%@ Register TagPrefix="apress" Namespace="ControlsBookLib.Ch05"
Assembly="ControlsBookLib" %>
<%@ Page language="c#" Codebehind="CustomEventTextbox.aspx.cs" AutoEventWireup="false"
Inherits="ControlsBookWeb.Ch05.CustomEventTextbox" %>
<%@ Register TagPrefix="apressUC" TagName="ControlsBookHeader"
Src="..\ControlsBookHeader.ascx" %>
<%@ Register TagPrefix="apressUC" TagName="ControlsBookFooter"
Src="..\ControlsBookFooter.ascx" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>Ch05 CustomEventTextbox</title>
    <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema" c
          ontent="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>
  <body MS_POSITIONING="FlowLayout">
    <form id="Textbox" method="post" runat="server">
      <apressUC:ControlsBookHeader id="Header"
           runat="server" ChapterNumber="5"
           ChapterTitle="Event-based Programming" />
      <h3>Ch05 CustomEventTextbox</h3>
      Enter your name:<br>
      <apress:customeventtextbox id="NameTextbox"
 runat="server"></apress:customeventtextbox><br>
      <br>
      <asp:button id="SubmitPageButton" runat="server"
              Text="Submit Page"></asp:button><br>
      <br>
      Before:<asp:label id="BeforeLabel" runat="server" Text=""></asp:label><br>
      After:<asp:label id="AfterLabel" runat="server" Text=""></asp:label><br>
      <apressUC:ControlsBookFooter id="Footer" runat="server" />
    </form>
  </body>
</HTML>
```

Listing 5-8. The CustomEventTextbox Web Form Code-Behind Class File

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
```

```
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ControlsBookWeb.Ch05
{
   public class CustomEventTextbox : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.Button SubmitPageButton;
      protected System.Web.UI.WebControls.Label BeforeLabel;
      protected System.Web.UI.WebControls.Label AfterLabel;
      protected ControlsBookLib.Ch05.CustomEventTextbox NameTextbox;

      private void Page_Load(object sender, System.EventArgs e)
      {
         BeforeLabel.Text = NameTextbox.Text;
         AfterLabel.Text = NameTextbox.Text;
      }

      #region Web Form Designer generated code
      override protected void OnInit(EventArgs e)
      {
         //
         // CODEGEN: This call is required by the ASP.NET Web Form Designer.
         //
         InitializeComponent();
         base.OnInit(e);
      }

      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.NameTextbox.TextChanged += new
        ControlsBookLib.Ch05.TextChangedEventHandler(this.NameCustom_TextChanged);
         this.Load += new System.EventHandler(this.Page_Load);

      }
      #endregion

      private void NameCustom_TextChanged(object o,
                ControlsBookLib.Ch05.TextChangedEventArgs tce)
      {
         BeforeLabel.Text = tce.OldValue;
         AfterLabel.Text = tce.NewValue;
      }
   }
}
```

We exercise the custom event by submitting the page by clicking the Submit Page button. This causes the `AfterLabel` control to change to "Dale Michalk," whereas the `BeforeLabel` keeps the old blank value, as shown in Figure 5-11.

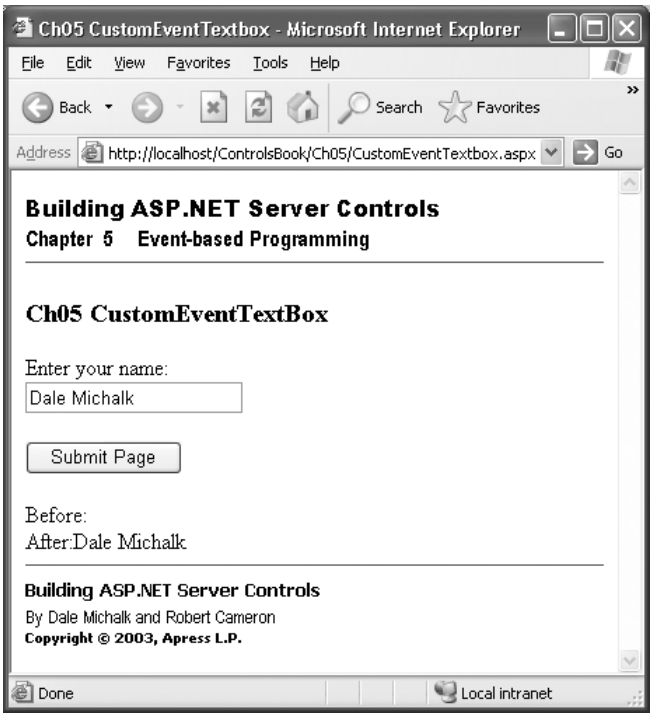

Figure 5-11. The page after submitting the CustomEventTextbox Web Form

The Visual Studio .NET Properties window did its job in wiring up to the custom event. It was smart enough to realize we had to use `TextChangedEventHandler` as a delegate to wrap the `NameCustom_TextChanged` event-handling method. This behavior by the Designer is one more reason we recommend sticking to the event model design pattern implemented in .NET. Here is the code that performs this work:

```
private void InitializeComponent()
{
   this.NameTextbox.TextChanged += new
        ControlsBookLib.Ch05.TextChangedEventHandler(this.NameCustom_TextChanged);
   this.Load += new System.EventHandler(this.Page_Load);
}
```

The following definition of `NameCustom_TextChanged` shows it is connected to `TextChanged` correctly, taking `TextChangedEventArgs` as its second parameter. The parameter named `tce` is the conduit to the information added to the `BeforeLabel` and `AfterLabel` `Text` values:

```
private void NameCustom_TextChanged(object o,
                      ControlsBookLib.Ch05.TextChangedEventArgs tce)
{
   BeforeLabel.Text = tce.OldValue;
   AfterLabel.Text = tce.NewValue;
}
```

Figure 5-12 shows what happens if we type a second name in the `CustomEventTextbox` control input box and click the Submit Page button to generate another postback. The control successfully remembers what the previous input was.

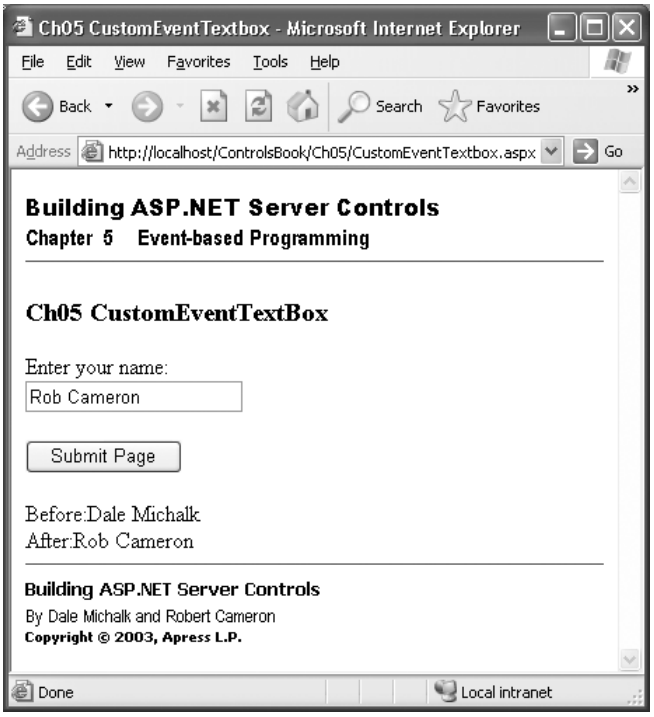

Figure 5-12. The second request with a new name on the CustomEventTextbox Web Form

Capturing Postback with the Button Control

The Textbox control does a great job in gathering input and raising state change events to their clients, but sometimes we need controls that provide action and post data back to the server. A perfect example of this type of control in the ASP.NET framework is the System.Web.UI.WebControls.Button control. The Button control exists for one reason: to post the page back to the server and raise events.

We would be remiss if we only reverse-engineered the ASP.NET TextBox control and left out the Button control, so our next task is to build our own version of the Button control. We add some bells and whistles along the way, such as the capability for the control to display itself as a Button or as a hyperlink similar to the LinkButton control in ASP.NET. This new, amazing Button server control will be named SuperButton for all its rich functionality.

Rendering the Button

The first decision we have to make when building our button relates to how it will render. Because we decided to render either as an <INPUT type="submit"> or an <A> tag, we choose to use a strongly-typed enumeration as a means to configure its display output. We call this enumeration ButtonDisplay and give it values that reflect how our button can appear in a Web Form:

```
public enum ButtonDisplay
{
   Button = 0,
   Hyperlink = 1
}
```

The ButtonDisplay enumeration is exposed from our control through a Display property. It defaults to a Button value if nothing is passed into the control:

```
public virtual ButtonDisplay Display
{
   get
   {
      object display = ViewState["Display"];
      if (display == null)
         return ButtonDisplay.Button;
      else
         return (ButtonDisplay) display;
   }
   set
   {
      ViewState["Display"] = value;
   }
}
```

We also have a `Text` property that has an identical representation in the code to our previous examples. It will appear as text on the surface of the button or as the text of the hyperlink.

The button-rendering code needs to have an if/then construct to switch the display based on the enumeration value set by the developer/user. It also needs a way to submit the page back to the web server when using the hyperlink display mode. The hyperlink is normally used for navigation and is not wired into the postback mechanism that buttons get for free.

The `Page` class comes to the rescue in this instance. It has a static method named `GetPostBackClientHyperlink` that registers the JavaScript necessary to submit the Web Form via an HTTP POST. In the Web Form example that hosts our `SuperButton` control, we examine the HTML output to see how it is integrated into the postback process. Here is the code that hooks into the postback mechanism:

```
override protected void Render(HtmlTextWriter writer)
{
   base.Render(writer);
   Page.VerifyRenderingInServerForm(this);

   if (Display == ButtonDisplay.Button)
   {
      writer.Write("<INPUT type=\"submit\"");
      writer.Write(" name=\"" + this.UniqueID + "\"");
      writer.Write(" id=\"" + this.UniqueID + "\"");
      writer.Write(" value=\"" + Text + "\"");
        writer.Write(" />");
   }
   else if (Display == ButtonDisplay.Hyperlink)
   {
      writer.Write("<A href=\"");
      writer.Write(Page.GetPostBackClientHyperlink(this,""));
      writer.Write("\">" + Text + "</A>");
   }
}
```

Exposing a Click Event and the Events Collection

The first event we add to our `SuperButton` control is a `Click` event. This is your garden-variety `System.EventHandler` delegate type event, but our actual event implementation will be different this time around. Instead of adding an event field to the control class, we reuse a mechanism given to all controls from the `System.Web.UI.Control` base class.

The `Events` read-only property inherited from the `Control` class provides access to an event collection of type `System.ComponentModel.EventHandlerList`. `EventHandlerList` provides access to delegates that represent the invocation list for each event the control exposes. This means that the only memory taken up to handle event delegates is by those events that have a client event handler method registered, unlike the previous technique, which takes a hit for each event, regardless of any clients using it. This can

potentially save a fair amount of memory on a control that exposes many events. Figure 5-13 graphically depicts the benefits of using the `Events` collection.

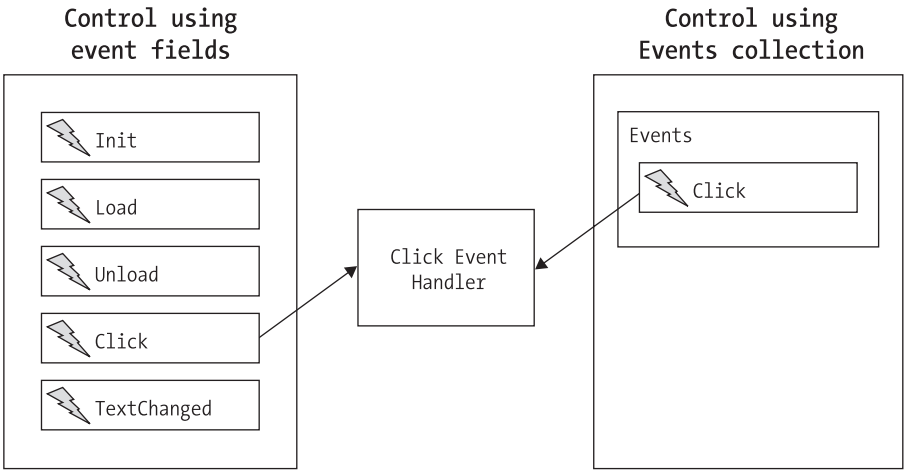

Figure 5-13. The difference between using an event field and using the `Events` *collection*

The first thing we need to do for an event using this new model is provide a key for the delegate that is used to store it inside the `Events` collection. We add this at the top of our class by creating a generic static, read-only object to represent the key for our click-related delegate:

```
private static readonly object ClickEvent = new object();
```

The second step is to use the syntax C# provides for custom delegate registration with our `Click` event. It is an expansion of the event declaration used previously that includes `add` and `remove` code blocks. It is similar to the `get` and `set` code blocks that programmers can use to define properties in C#. The result is the following `Click` event:

```
public event EventHandler Click
{
   add
   {
      Events.AddHandler(ClickEvent, value);
   }
   remove
   {
      Events.RemoveHandler(ClickEvent, value);
   }
}
```

The first thing to notice is the event declaration itself. It is declared with an `event` keyword, delegate type, name, and accessibility modifier as before. The new functionally is added via code blocks below the declaration. The `add` and `remove` code blocks handle the delegate registration process in whatever manner they see fit to do so. In this case, these code blocks are passed the delegate reference via the `value` keyword to accomplish their assigned tasks.

The code in our `Click` event uses the `Events` collection to add the delegate via `AddHandler` or to remove the delegate via `RemoveHandler`. `ClickEvent` is the access key used to identify the `Click` delegates in our `Events` collection, keeping like event handlers in separate buckets.

After we declare our event with its event subscription code, we need to define our `OnClick` method to raise the event. The code uses the `Events` collection and our defined key object to get the `Click` delegate and raise the event to subscribers:

```
protected virtual void OnClick(EventArgs e)
{
   EventHandler clickEventDelegate = (EventHandler)Events[ClickEvent];
   if (clickEventDelegate != null)
   {
      clickEventDelegate(this, e);
   }
}
```

The first step is to pull the delegate of type `EventHandler` from the `Events` collection. Our second step as before is to check it for a `null` value to ensure that we actually need to invoke it. The invocation code on the delegate is the same as we used previously with our event in the `Textbox` demonstrations. We invoke the delegate using function call syntax with the name of the delegate. At this point, our `Click` event is ready to go—all we need to do is raise it when a postback occurs.

Command Events and Event Bubbling

The second event exposed by our `SuperButton` control is a *command* event. The command event is a design pattern borrowed from the controls in the `System.Web.UI.WebControls` namespace that makes event handling in list controls easier.

The primary example of this scenario is the `DataGrid` control, which can have buttons embedded in a column for edit and delete operations. The buttons activate edit or delete functionality respectively in the `DataGrid` control as long as the command events exposed by these buttons have the correct `CommandName` property in the `CommandEventArgs` class as part of the event. If the button is set with a `CommandName` of "Delete", it kicks off delete activity. If the button is set with a `CommandName` of "Edit", it starts edit functions in the `DataGrid` control. Controls that raise command events that are not in those expected by the `DataGrid` control are wrapped into an `ItemCommand` event exposed by the control.

The capabilities provided by a command event are an implementation of event bubbling. *Event bubbling* is a technique that allows a child control to propagate command events up its control hierarchy, allowing the event to be handled in a more convenient location. Figure 5-14 provides a graphical depiction of event bubbling. This technique allows the `DataGrid` control to take a crack at handling the button events despite the fact that the buttons are several layers deep inside of its control hierarchy.

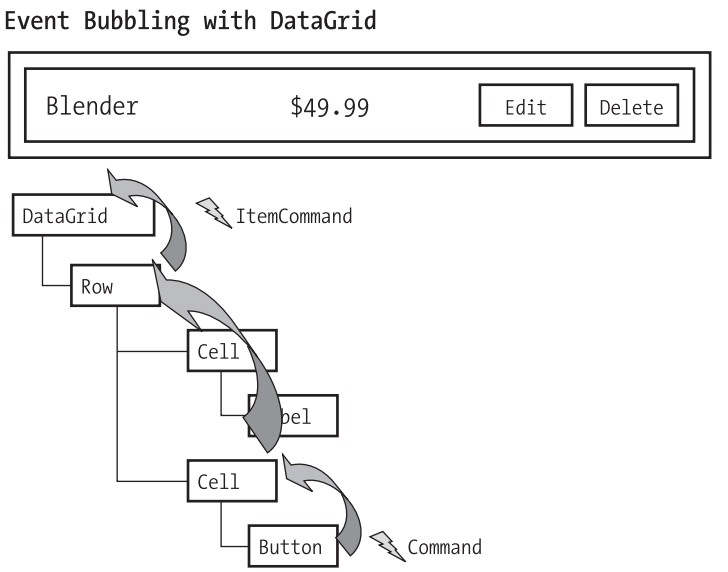

Figure 5-14. Event bubbling

Exposing the Command Event

The techniques used to expose a command event on our control are similar to those used with the `Click` event. As before, an important preliminary task to creating the event declaration is the need for an object to provide a "key" that gives access to the event in the `Events` collection. The `CommandEvent` field handles this chore:

```
private static readonly object CommandEvent = new object();
```

The event declaration for the `Command` event is almost identical to the `Click` event except for the delegate type used. It exposes the `CommandEventHandler` delegate, which provides data through the `CommandEventArgs` parameter to clients registered to process the event:

```
public event CommandEventHandler Command
{
   add
   {
      Events.AddHandler(CommandEvent, value);
   }
   remove
   {
      Events.RemoveHandler(CommandEvent, value);
   }
}
```

The `CommandEventArgs` class provides two properties: `CommandName` and `CommandArgument`. A control is expected to maintain these values as part of a command event bubbling protocol. These values are copied directly into the `CommandEventArgs` class when the command event is raised. Command controls expose these values through the `CommandName` and `CommandArgument` public properties, respectively:

```
public virtual string CommandName
{
   get
   {
      object name = ViewState["CommandName"];
      if (name == null)
         return string.Empty;
      else
         return (string) name;
   }
   set
   {
      ViewState["CommandName"] = value;
   }
}

public virtual string CommandArgument
{
   get
   {
      object arg = ViewState["CommandArgument"];
      if (arg == null)
         return string.Empty;
      else
         return (string) arg;
   }
   set
   {
      ViewState["CommandArgument"] = value;
   }
}
```

The final step in working with a command event is to raise the event. The `OnCommand` method in our class holds this important code. It pulls back the appropriate delegate type from the `Events` collection and invokes it in a similar manner to the `OnClick` method we reviewed earlier:

```
protected virtual void OnCommand(CommandEventArgs ce)
{
   CommandEventHandler commandEventDelegate =
      (CommandEventHandler) Events[CommandKey];
   if (commandEventDelegate != null)
   {
      commandEventDelegate(this, ce);
   }

   RaiseBubbleEvent(this, ce);
}
```

The new code that stands out is the `RaiseBubbleEvent` method call at the end of the `OnCommand` method. This code takes advantage of the internal event-bubbling plumbing that all controls receive just by inheriting from `System.Web.UI.Control`.

`RaiseBubbleEvent` takes an object reference and a `System.EventArgs` reference for its two parameters. This permits all events, even those not related to command event functionality, to take advantage of event bubbling. Naturally, the primary concern of event bubbling in ASP.NET is with command events.

At this point in our design, we have successfully exposed both the `Click` event and the command event for our control using the `Events` collection. One of the limitations of the `Events` collection is its implementation as a linked list. Given the nature of the linked list data structure, it can cause a performance problem in certain scenarios when many delegate nodes are traversed in order to find the correct event delegate. As background, you are free to use other collection types to hold event delegates. One alternative to using a linked list is to implement the events collection as a `Hashtable`, which can speed access.

Capturing the Postback via IPostBackEventHandler

As part of our design, we had the requirement of rendering the button as either a normal button or as a specially configured hyperlink to submit the Web Form. With events in hand, we now move on to hooking the button click into the postback process through implementation of the `IPostBackEventHandler` interface. To achieve this, we next implement the single method of the postback interface, `RaisePostBackEvent`:

```
public void RaisePostBackEvent(string argument);
```

RaisePostBackEvent takes a single argument as a means to retrieve a value from the form submission. When a Button submits a Web Form, it always passes a blank value for this argument to RaisePostBackEvent. Our hyperlink-rendering code has a choice of what information to pass via the Page.GetPostBackClientHyperlink method call. The following code snippet submits a blank value to keep things in line with our button rendering:

```
writer.Write("<A href=\"");
writer.Write(Page.GetPostBackClientHyperlink(this,""));
writer.Write("\">" + Text + "</A>");
```

The RaisePostBackEvent implementation in our SuperButton control has very little work to do, as we encapsulated the bulk of our event-generating code in the OnClick and OnCommand methods:

```
public void RaisePostBackEvent(string argument)
{
   OnCommand(new CommandEventArgs(CommandName, CommandArgument));
   OnClick(EventArgs.Empty);
}
```

Completing the RaisePostBackEvent method brings our SuperButton control to fruition. Listing 5-9 is the class file for the control and its related enumeration. The control needs a "using" import for the System.Web.UI.WebControls namespace because it takes advantage of Command events.

Listing 5-9. The SuperButton *Control Class File*

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace ControlsBookLib.Ch05
{
   public enum ButtonDisplay
   {
      Button = 0,
      Hyperlink = 1
   }

   [ToolboxData("<{0}:SuperButton runat=server></{0}:SuperButton>")]
   public class SuperButton : Control, IPostBackEventHandler
   {
      public virtual ButtonDisplay Display
      {
         get
         {
            object display = ViewState["Display"];
```

```
      if (display == null)
         return ButtonDisplay.Button;
      else
         return (ButtonDisplay) display;
   }
   set
   {
      ViewState["Display"] = value;
   }
}

public virtual string Text
{
   get
   {
      object text = ViewState["Text"];
      if (text == null)
         return string.Empty;
      else
         return (string) text;
   }
   set
   {
      ViewState["Text"] = value;
   }
}

private static readonly object ClickKey = new object();

public event EventHandler Click
{
   add
   {
      Events.AddHandler(ClickKey, value);
   }
   remove
   {
      Events.RemoveHandler(ClickKey, value);
   }
}

protected virtual void OnClick(EventArgs e)
{
   EventHandler clickEventDelegate =
      (EventHandler)Events[ClickKey];
   if (clickEventDelegate != null)
   {
      clickEventDelegate(this, e);
   }
}
```

```
private static readonly object CommandKey = new object();

public event CommandEventHandler Command
{
   add
   {
      Events.AddHandler(CommandKey, value);
   }
   remove
   {
      Events.RemoveHandler(CommandKey, value);
   }
}

public virtual string CommandName
{
   get
   {
      object name = ViewState["CommandName"];
      if (name == null)
         return string.Empty;
      else
         return (string) name;
   }
   set
   {
      ViewState["CommandName"] = value;
   }
}

public virtual string CommandArgument
{
   get
   {
      object arg = ViewState["CommandArgument"];
      if (arg == null)
         return string.Empty;
      else
         return (string) arg;
   }
   set
   {
      ViewState["CommandArgument"] = value;
   }
}

protected virtual void OnCommand(CommandEventArgs ce)
{
   CommandEventHandler commandEventDelegate =
      (CommandEventHandler) Events[CommandKey];
   if (commandEventDelegate != null)
```

```
            {
               commandEventDelegate(this, ce);
            }

            RaiseBubbleEvent(this, ce);
         }

         public void RaisePostBackEvent(string argument)
         {
            OnCommand(new CommandEventArgs(CommandName, CommandArgument));
            OnClick(EventArgs.Empty);
         }

         override protected void Render(HtmlTextWriter writer)
         {
            base.Render(writer);
            Page.VerifyRenderingInServerForm(this);

            if (Display == ButtonDisplay.Button)
            {
               writer.Write("<INPUT type=\"submit\"");
               writer.Write(" name=\"" + this.UniqueID + "\"");
               writer.Write(" id=\"" + this.UniqueID + "\"");
               writer.Write(" value=\"" + Text + "\"");
               writer.Write(" />");
            }
            else if (Display == ButtonDisplay.Hyperlink)
            {
               writer.Write("<A href=\"");
               writer.Write(Page.GetPostBackClientHyperlink(this,""));
               writer.Write("\">" + Text + "</A>");
            }
         }
      }
   }
```

Using the SuperButton Control on a Web Form

The SuperButton Web Form hosts two `SuperButton` controls: one of the button variety and the other of the hyperlink persuasion. It also has a label that is set according to event handlers for each button. The first request to the Web Form generates Figure 5-15. Listings 5-10 and 5-11 provide the source code for this Web Form.

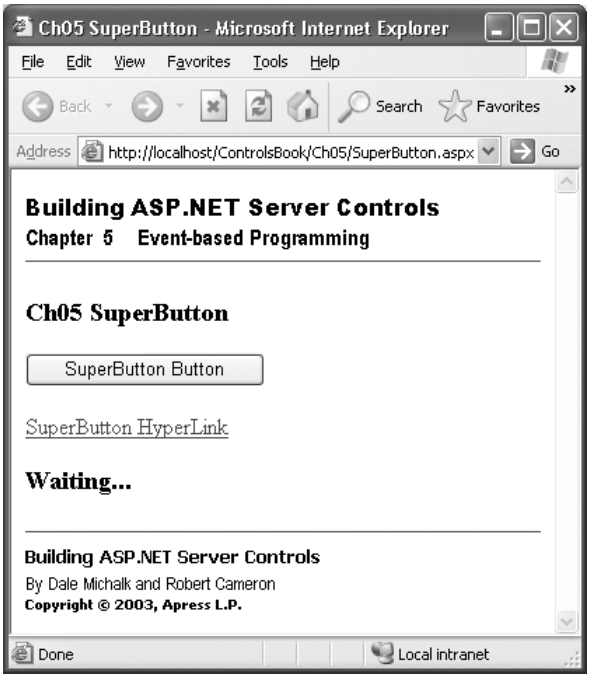

Figure 5-15. The SuperButton Web Form rendering its first request

Listing 5-10. The SuperButton Web Form .aspx Page File

```
<%@ Page language="c#" Codebehind="SuperButton.aspx.cs" AutoEventWireup="false"
Inherits="ControlsBookWeb.Ch05.SuperButton" %>
<%@ Register TagPrefix="apress" Namespace="ControlsBookLib.Ch05"
Assembly="ControlsBookLib" %>
<%@ Register TagPrefix="apressUC" TagName="ControlsBookHeader"
Src="..\ControlsBookHeader.ascx" %>
<%@ Register TagPrefix="apressUC" TagName="ControlsBookFooter"
Src="..\ControlsBookFooter.ascx" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>Ch05 SuperButton</title>
    <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
    content="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>
  <body MS_POSITIONING="FlowLayout">
    <form id="SuperButton" method="post" runat="server">
      <apressUC:ControlsBookHeader id="Header" runat="server" ChapterNumber="5"
```

```
ChapterTitle="Event-based Programming" />
      <h3>Ch05 SuperButton</h3>
      <apress:superbutton id="superbtn" runat="server" Text="SuperButton
Button"></apress:superbutton><br>
      <br>
      <apress:superbutton display="hyperlink" id="superlink" runat="server"
Text="SuperButton HyperLink"></apress:superbutton><br>
      <br>
      <h3><asp:Label id="ClickLabel" runat="server">Waiting...</asp:Label></h3>
      <apressUC:ControlsBookFooter id="Footer" runat="server" />
    </form>
  </body>
</HTML>
```

Listing 5-11. The SuperButton Web Form Code-Behind Class File

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ControlsBookWeb.Ch05
{
   public class SuperButton : System.Web.UI.Page
   {
      protected ControlsBookLib.Ch05.SuperButton superlink;
      protected System.Web.UI.WebControls.Label ClickLabel;
      protected ControlsBookLib.Ch05.SuperButton superbtn;

      private void Page_Load(object sender, System.EventArgs e)
      {
         ClickLabel.Text = "Waiting...";
      }

      #region Web Form Designer generated code
      override protected void OnInit(EventArgs e)
      {
         //
         // CODEGEN: This call is required by the ASP.NET Web Form Designer.
         //
         InitializeComponent();
         base.OnInit(e);
      }

      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
```

```
        /// </summary>
        private void InitializeComponent()
        {
            this.superbtn.Click += new System.EventHandler(this.superbtn_Click);
            this.superlink.Click += new System.EventHandler(this.superlink_Click);
            this.Load += new System.EventHandler(this.Page_Load);

        }
        #endregion

        private void superbtn_Click(object sender, System.EventArgs e)
        {
            ClickLabel.Text = "superbtn was clicked!";
        }

        private void superlink_Click(object sender, System.EventArgs e)
        {
            ClickLabel.Text = "superlink was clicked!";
        }
    }
}
```

Clicking the button generates the output in Figure 5-16. Clicking the hyperlink generates the output in Figure 5-17.

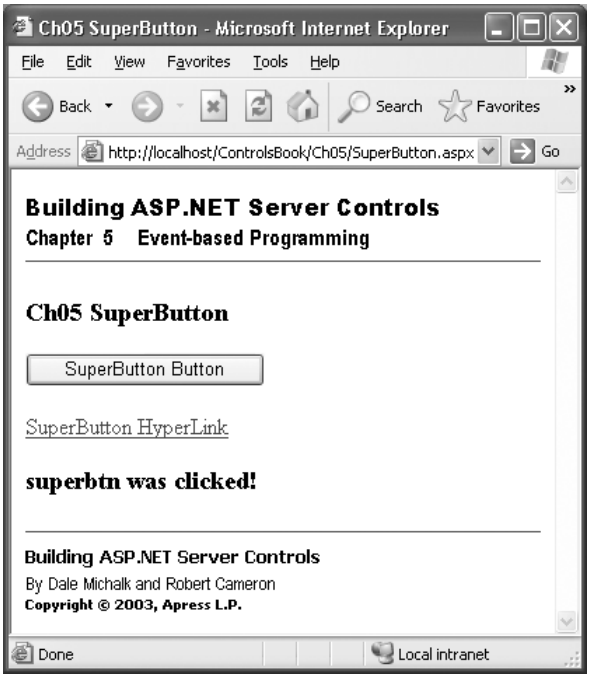

Figure 5-16. The SuperButton Web Form after a button click

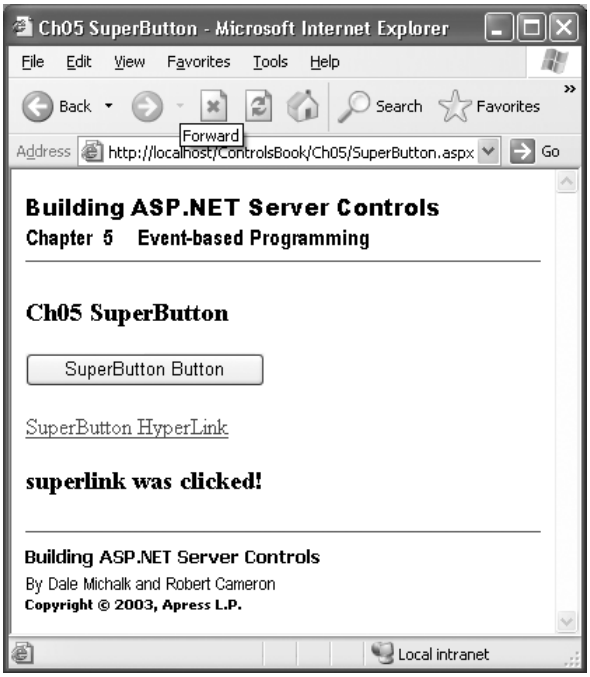

Figure 5-17. The SuperButton Web Form after a hyperlink click

Of more interest is what is rendered on the HTML page that represents the Web Form. Listing 5-12 shows the HTML.

Listing 5-12. The SuperButton Web Form Rendered HTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
<HEAD>
   <title>Ch05 Super Button</title>
   <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
   <meta name="CODE_LANGUAGE" Content="C#">
   <meta name="vs_defaultClientScript" content="JavaScript">
   <meta name="vs_targetSchema"
   content="http://schemas.microsoft.com/intellisense/ie5">
</HEAD>
<body MS_POSITIONING="FlowLayout">
<form name="SuperButton" method="post" action="SuperButton.aspx" id="SuperButton">
<input type="hidden" name="__VIEWSTATE"
value="dDwtMTg0MzczNjQyMztOPDtsPGk8MT47PjtsPHQ8O2w8aTw1Pjs+O2w8dDxwPHA8bDxUZXhOOz4
7bDxzdXBlcmxpbmsgd2FzIGNsaWNrZWQhOz4+Oz47Oz47
Pj47Pj47PiynDcUVBayIzW4ijbIz/Ks9mMxm" />
   <h3>Ch05 SuperButton</h3>
   <INPUT type="submit" name="superbtn" id="superbtn"
    value="SuperButton Button" /><br>
   <br>
```

```
   <A href="javascript:__doPostBack('superlink','')">SuperButton HyperLink</A><br>
   <br>
   <h3><span id="ClickLabel">superlink was clicked!</span></h3>

<input type="hidden" name="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" value="" />
<script language="javascript">
<!--
      function __doPostBack(eventTarget, eventArgument) {
         var theform = document.SuperButton;
         theform.__EVENTTARGET.value = eventTarget;
         theform.__EVENTARGUMENT.value = eventArgument;
         theform.submit();
      }
// -->
</script>
</form>
</body>
</HTML>
```

The first thing to examine is how our hyperlink generates a postback:

```
<A href="javascript:__doPostBack('superlink','')">SuperButton HyperLink</A><br>
```

It uses a JavaScript function named `__doPostBack`, which actually sends the page back to the server. This JavaScript invocation is added by our `Page.GetPostBackClientHyperlink` call in the `Render` method of `SuperButton`. The `__doPostBack` JavaScript routine is emitted into the HTML by the ASP.NET framework as a result of this method call:

```
<input type="hidden" name="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" value="" />
<script language="javascript">
<!--
      function __doPostBack(eventTarget, eventArgument) {
         var theform = document.SuperButton;
         theform.__EVENTTARGET.value = eventTarget;
         theform.__EVENTARGUMENT.value = eventArgument;
         theform.submit();
      }
// -->
</script>
```

The JavaScript code programmatically submits the form and sets two hidden variables to give ASP.NET enough information about what control was responsible for causing the postback. It doesn't need this extra step when rendering the `<INPUT type="submit">` button, but it is mandatory for hyperlinks. You can also see that the purpose of the second parameter in `Page.GetPostBackClientHyperlink` is to pass an `eventArgument`, which makes its way back to the `RaisePostBack` method invocation on the server-side control implementation as the string parameter named `argument`.

Composing the SuperButton Control into a Composite Pager Control

Our SuperButton control is capable of raising command events through the event-bubbling mechanism. To capture these bubbled events, we use a composite control named Pager. Pager recognizes bubbled command events from its children and raises a PageCommand event to its event clients. This is similar to the event bubbling performed by the DataGrid list control when it grabs all command events from child controls and exposes them via a single ItemCommand event. We next describe the design of the Pager control, starting with how the control is constructed.

Building the Pager Child Control Hierarchy

Composite control development begins with creating a child control hierarchy. The Pager control uses a private method named CreateChildControlHierarchy that is called from the overridden protected CreateChildControls method. Listing 5-13 provides the source code for CreateChildControlHierarchy. CreateChildControls is called by the ASP.NET framework to allow composite controls to build up their structure prior to rendering.

Listing 5-13. The Pager *Implementation of* CreateChildControlHierarchy

```
private SuperButton buttonLeft ;
private SuperButton buttonRight;
private void CreateChildControlHierarchy()
{
   LiteralControl tableStart = new
      LiteralControl("<table border=1><tr><td>");
   Controls.Add(tableStart);

      buttonLeft = new SuperButton();
   buttonLeft.ID = "buttonLeft";
   if (Context != null)
   {
      buttonLeft.Text = Context.Server.HtmlEncode("<") + " Left";
   }
   else
   {
      buttonLeft.Text = "< Left";
   }
   buttonLeft.CommandName = "Page";
   buttonLeft.CommandArgument = "Left";
   Controls.Add(buttonLeft);

   LiteralControl spacer = new LiteralControl("  ");
   Controls.Add(spacer);

   buttonRight = new SuperButton();
```

```
   buttonRight.ID = "buttonRight";
   buttonRight.Display = Display;
   if (Context != null)
   {
      buttonRight.Text = "Right " + Context.Server.HtmlEncode(">");
   }
   else
   {
      buttonRight.Text = "Right  >";
   }
   buttonRight.CommandName = "Page";
   buttonRight.CommandArgument = "Right";
   Controls.Add(buttonRight);

   LiteralControl tableEnd = new
      LiteralControl("</td></tr></table>");
   Controls.Add(tableEnd);
}
```

The child control collection created by the `Pager` control includes a set of `SuperButtons` representing left and right direction arrows that are wrapped inside an HTML table. The `Left` direction `SuperButton` includes the text `< Left`, and the `Right` direction `SuperButton` uses `Right >`. The `Text` property uses `HtmlEncode` to properly render the special characters. Otherwise, `CreateChildControlHierarchy` renders straight text when `Context` is not available at design time.

```
if (Context != null)
{
   buttonLeft.Text = Context.Server.HtmlEncode("<") + " Left";
}
else
{
   buttonLeft.Text = "< Left";
}
```

The most important settings in `CreateChildControlHierarchy` are the `Command` properties. The `CommandName` value chosen for the `SuperButton` controls is `Page`. This lets the `Pager` know that it is receiving `Command` events from its specially configured `SuperButton` controls. `CommandArgument` tells the `Pager` whether it is the left or right control emitting the event:

```
buttonLeft.CommandName = "Page";
buttonLeft.CommandArgument = "Left";
…
buttonRight.CommandName = "Page";
buttonRight.CommandArgument = "Right";
```

The final rendering feature is the Display property passed on to the SuperButton controls. Our Pager can display its left and right UI elements as either buttons or hyperlinks. The implementation of the Display property in Pager is as follows. It calls EnsureChildControls and then gets or sets the Display property on the child controls. The SuperButton server control defaults to a Display value of Button, which becomes the default for Pager as well if the value is not set.

```
public virtual ButtonDisplay Display
{
   get
   {
      EnsureChildControls();
      return buttonLeft.Display ;
   }
   set
   {
      EnsureChildControls();
      buttonLeft.Display  = value;
      buttonRight.Display = value;
   }
}
```

Defining the PageCommand Event

The Pager control exposes a custom PageCommand event to let its client know whether it is moving in the left or right direction. The PageDirection enumeration provides a finite way to specify this in code:

```
public enum PageDirection
{
   Left = 0,
   Right = 1
}
```

The PageCommandEventArgs class uses this enumeration as the data type for its Direction property exposed as part of an EventArgs replacement for the PageCommand delegate. The complete PageCommand event–related code is grouped in the PageCommand class file shown in Listing 5-14.

Listing 5-14. The PageCommand *Class File*

```
using System;

namespace ControlsBookLib.Ch05
{
   public enum PageDirection
   {
      Left = 0,
      Right = 1
   }
```

```
   public delegate void PageCommandEventHandler(object o,
                                          PageCommandEventArgs pce);

   public class PageCommandEventArgs
   {
      public PageCommandEventArgs(PageDirection direction)
      {
         this.direction = direction;
      }

      PageDirection direction;
      public PageDirection Direction
      {
         get{ return direction; }
      }
   }
}
```

Exposing the PageCommand Event from the Pager Control

The `Pager` control uses the `PageCommandEventHandler` delegate to declare its event-handling code. As with the `SuperButton`, we use the `Events` property technique for handling delegate registration:

```
private static readonly object PageCommandKey = new object();
public event PageCommandEventHandler PageCommand
{
   add
   {
      Events.AddHandler(PageCommandKey, value);
   }
   remove
   {
      Events.RemoveHandler(PageCommandKey, value);
   }
}
```

We also add an `OnPageCommand` method to raise the event. This method uses the custom `PageCommandEventArgs` class we defined earlier to invoke the `PageCommandEventHandler` delegate:

```
protected virtual void OnPageCommand(PageCommandEventArgs pce)
{
   PageCommandEventHandler pageCommandEventDelegate =
               (PageCommandEventHandler) Events[PageCommandEvent];
   if (pageCommandEventDelegate != null)
   {
      pageCommandEventDelegate(this, pce);
   }
}
```

OnPageCommand is the last bit of code required to raise events associated with the PageCommand event type. The next task is to capture the bubbled Command events and turn them into PageCommand events.

Capturing the Bubbles via OnBubbleEvent

The OnBubbleEvent method inherited from System.Web.UI.Control is the counterpart to the RaiseBubbleEvent method used inside the SuperButton control. It allows a control to hook into the stream of bubbled events from child controls and process them accordingly:

```
protected override bool OnBubbleEvent(object source, EventArgs e);
```

The method definition for OnBubbleEvent specifies the ubiquitous System.EventHandler method signature, with one difference. It takes an object reference and an EventArgs reference, but returns a bool. The bool return value indicates whether or not the control has processed the bubble event. A value of false indicates that the bubble event should continue bubbling up the control hierarchy; a value of true indicates a desire to stop the event in its tracks because it has been handled. If a control does not implement OnBubbleEvent, the default implementation passes the event on up to parent controls.

The Pager control implements its OnBubbleEvent as shown in Listing 5-15.

Listing 5-15. The Pager *Implementation of* OnBubbleEvent

```
protected override bool OnBubbleEvent(object source, EventArgs e)
{
   bool result = false;
   CommandEventArgs ce = e as CommandEventArgs;

   if (ce != null)
   {
      if (ce.CommandName.Equals("Page"))
      {
         PageDirection direction;
         if (ce.CommandArgument.Equals("Right"))
            direction = PageDirection.Right;
         else
            direction = PageDirection.Left;

         PageCommandEventArgs pce =
            new PageCommandEventArgs(direction);

         OnPageCommand(pce);
         result = true;
      }
   }
   return result;
}
```

The result variable holds the return value of OnBubbleEvent for the Pager control. It is set to false, assuming failure until success. The first check is to cast the EventArgs reference to ensure we receive a Command event of the proper type. The code performs this check using the as keyword in C# to cast the reference to the desired type, which returns null if the cast fails.

If the type cast succeeds, the next check is to ensure the proper CommandName is set to "Page". After the checks pass, the OnBubbleEvent code can create a PageCommandEventArgs class and set the Direction property according to the CommandArgument value. The final task is to raise the PageCommand event by calling OnPageCommand. Finally, the function returns the value of result to tell the ASP.NET framework whether or not the event was handled.

The INamingContainer Interface

When a composite control builds up its child control tree, it sets each control's identification via the ID property. For example, the Pager control sets the left SuperButton child control ID property value in the following single line of code:

```
buttonLeft.ID = "buttonLeft";
```

The problem with using just the ID value to uniquely identify child controls is that multiple Pager controls could be used on a Web Form, and the emitted button or hyperlink ID values would conflict. To protect against name collisions, each composite control creates a unique namespace that prefixes the ID of a control with the parent control's ID and a colon. The INamingContainer interface tells ASP.NET to do this. INamingContainer is a marker interface (i.e., an interface without any defined methods) used by ASP.NET to identify the parent in a composite control to ensure unique names or IDs for child controls as they are dynamically created during the page-rendering process.

Implementing the INamingContainer interface in the Pager server control activates this mechanism, causing ASP.NET to prefix the ID of a control with the parent control's ID and a colon. The previous left button in a Pager control named "pagerbtn" would therefore have an ID value of "buttonLeft" but a UniqueID value of "pagerbtn:buttonLeft". Listing 5-16 contains the full code listing for the Pager control.

Listing 5-16. The Pager *Control Class File*

```
using System;
using System.ComponentModel;
using System.Web.UI;
using System.Web.UI.WebControls;
using ControlsBookLib.Ch12.Design;

namespace ControlsBookLib.Ch05
{
   [ToolboxData("<{0}:Pager
runat=server></{0}:Pager>"),Designer(typeof(CompCntrlDesigner))]
   public class Pager : Control, INamingContainer
```

```
{
   private static readonly object PageCommandKey = new object();
   public event PageCommandEventHandler PageCommand
   {
      add
      {
         Events.AddHandler(PageCommandKey, value);
      }
      remove
      {
         Events.RemoveHandler(PageCommandKey, value);
      }
   }

   protected virtual void OnPageCommand(PageCommandEventArgs pce)
   {
      PageCommandEventHandler pageCommandEventDelegate =
         (PageCommandEventHandler) Events[PageCommandKey];
      if (pageCommandEventDelegate != null)
      {
         pageCommandEventDelegate(this, pce);
      }
   }

   protected override bool OnBubbleEvent(object source, EventArgs e)
   {
      bool result = false;
      CommandEventArgs ce = e as CommandEventArgs;

      if (ce != null)
      {
         if (ce.CommandName.Equals("Page"))
         {
            PageDirection direction;
            if (ce.CommandArgument.Equals("Right"))
               direction = PageDirection.Right;
            else
               direction = PageDirection.Left;

            PageCommandEventArgs pce =
               new PageCommandEventArgs(direction);

            OnPageCommand(pce);
            result = true;
         }
      }
      return result;
   }

   public virtual ButtonDisplay Display
   {
      get
      {
         EnsureChildControls();
         return buttonLeft.Display ;
      }
```

```
   set
   {
      EnsureChildControls();
      buttonLeft.Display  = value;
      buttonRight.Display = value;
   }
}

protected override void CreateChildControls()
{
   Controls.Clear();
   CreateChildControlHierarchy();
}

public override ControlCollection Controls
{
   get
   {
      EnsureChildControls();
      return base.Controls;
   }
}

private SuperButton buttonLeft ;
private SuperButton buttonRight;
private void CreateChildControlHierarchy()
{
   LiteralControl tableStart = new
      LiteralControl("<table border=1><tr><td>");
   Controls.Add(tableStart);

    buttonLeft = new SuperButton();
   buttonLeft.ID = "buttonLeft";
   if (Context != null)
   {
      buttonLeft.Text = Context.Server.HtmlEncode("<") + " Left";
   }
   else
   {
      buttonLeft.Text = "< Left";
   }
   buttonLeft.CommandName = "Page";
   buttonLeft.CommandArgument = "Left";
   Controls.Add(buttonLeft);

   LiteralControl spacer = new LiteralControl("  ");
   Controls.Add(spacer);

   buttonRight = new SuperButton();
   buttonRight.ID = "buttonRight";
   buttonRight.Display = Display;
   if (Context != null)
   {
      buttonRight.Text = "Right " + Context.Server.HtmlEncode(">");
   }
   else
   {
```

```
            buttonRight.Text = "Right  >";
         }
         buttonRight.CommandName = "Page";
         buttonRight.CommandArgument = "Right";
         Controls.Add(buttonRight);

         LiteralControl tableEnd = new
            LiteralControl("</td></tr></table>");
         Controls.Add(tableEnd);
      }
   }
}
```

Using the Pager Control on a Web Form

The Pager Event Bubbling Web Form demonstrates the `Pager` control in both its button and hyperlink display motifs. A single label represents the `PageCommand` activity generated by the two controls. The first request for the page appears in the browser, as shown in Figure 5-18. Listings 5-17 and 5-18 provide the .aspx and code-behind files for this Web Form.

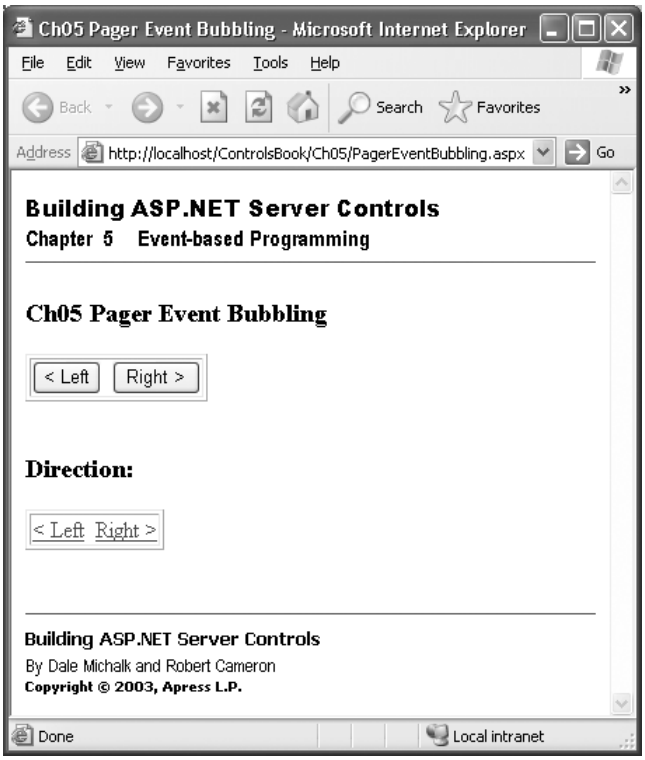

Figure 5-18. The Pager Event Bubbling Web Form rendering its first request

Listing 5-17. The Pager Event Bubbling Web Form .aspx Page File

```
<%@ Register TagPrefix="apress" Namespace="ControlsBookLib.Ch05"
Assembly="ControlsBookLib" %>
<%@ Page language="c#" Codebehind="PagerEventBubbling.aspx.cs"
AutoEventWireup="false"
Inherits="ControlsBookWeb.Ch05.PagerEventBubbling" %>
<%@ Register TagPrefix="apressUC" TagName="ControlsBookHeader"
Src="..\ControlsBookHeader.ascx" %>
<%@ Register TagPrefix="apressUC" TagName="ControlsBookFooter"
Src="..\ControlsBookFooter.ascx" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>Ch05 Pager Event Bubbling</title>
    <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
            content="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>
  <body MS_POSITIONING="FlowLayout">
    <form id="PagerEventBubbling" method="post" runat="server">
      <apressUC:ControlsBookHeader id="Header" runat="server" ChapterNumber="5"
             ChapterTitle="Event-based Programming" />
      <h3>Ch05 Pager Event Bubbling</h3>
      <apress:pager id="pager1" display="button" runat="server"></apress:pager><br>
      <br>
      <h3>Direction: <asp:Label ID="DirectionLabel"
                  Runat="server"></asp:Label></h3>
      <apress:pager id="pager2" display="hyperlink"
                  runat="server"></apress:pager><br>
      <br>
      <apressUC:ControlsBookFooter id="Footer" runat="server" />
    </form>
  </body>
</HTML>
```

Listing 5-18. The Pager Event Bubbling Web Form Code-Behind Class File

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ControlsBookWeb.Ch05
{
   public class PagerEventBubbling : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.Label DirectionLabel;
```

```
      protected ControlsBookLib.Ch05.Pager pager1;
      protected ControlsBookLib.Ch05.Pager pager2;

      private void Page_Load(object sender, System.EventArgs e)
      {
         DirectionLabel.Text = "<none>";
      }

      #region Web Form Designer generated code
      override protected void OnInit(EventArgs e)
      {
         //
         // CODEGEN: This call is required by the ASP.NET Web Form Designer.
         //
         InitializeComponent();
         base.OnInit(e);
      }

      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.pager1.PageCommand +=
        new ControlsBookLib.Ch05.PageCommandEventHandler(this.Pagers_PageCommand);
         this.pager2.PageCommand +=
        new ControlsBookLib.Ch05.PageCommandEventHandler(this.Pagers_PageCommand);
         this.Load += new System.EventHandler(this.Page_Load);

      }
      #endregion

      private void Pagers_PageCommand(object o,
                                ControlsBookLib.Ch05.PageCommandEventArgs pce)
      {
         DirectionLabel.Text =
            Enum.GetName(typeof(ControlsBookLib.Ch05.PageDirection),
            pce.Direction);
      }
   }
}
```

The `Pager` controls are wired to the same event handler in the code-behind class named `Pagers_PageCommand` in the `InitializeComponent` method of the Web Form:

```
private void InitializeComponent()
{
   this.pager1.PageCommand += new
        ControlsBookLib.Ch05.PageCommandEventHandler(this.Pagers_PageCommand);
   this.pager2.PageCommand += new
        ControlsBookLib.Ch05.PageCommandEventHandler(this.Pagers_PageCommand);
   this.Load += new System.EventHandler(this.Page_Load);
}
```

`Pagers_PageCommand` has an all-important second parameter of type `PageCommandEventArgs`. We use it along with the `System.Enum` class's static `GetName` method to produce a textual representation of the `PageDirection` enumeration value for display in the `DirectionLabel` `Text` property:

```
private void Pagers_PageCommand(object o, C
                        ontrolsBookLib.Ch05.PageCommandEventArgs pce)
{
   DirectionLabel.Text =
      Enum.GetName(typeof(ControlsBookLib.Ch05.PageDirection),
               pce.Direction);
}
```

Click the Left button of the top `Pager` control to verify that it is working. The result should look something like Figure 5-19.

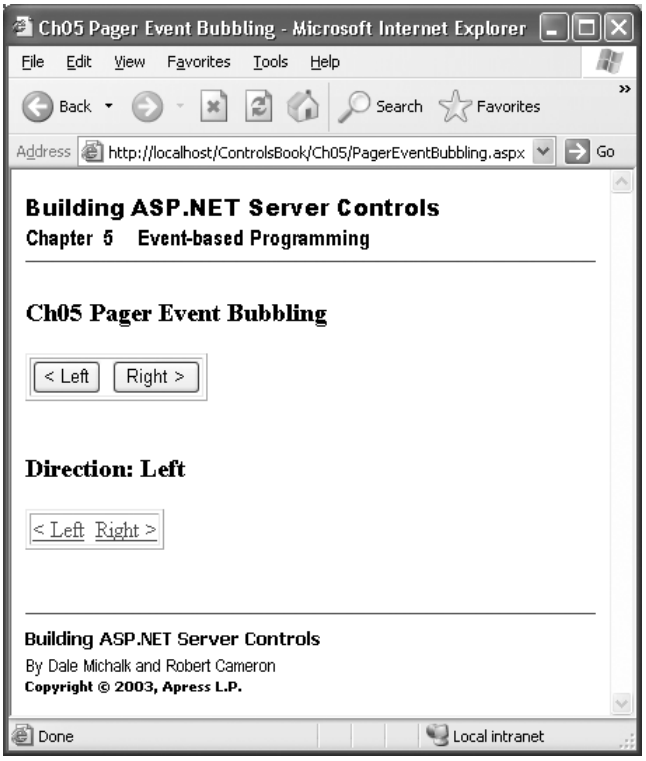

Figure 5-19. The Page Event Bubbling Web Form after clicking the Left hyperlink button

Try the Right button with the bottom Pager that is in a hyperlink form and you should get output similar to Figure 5-20.

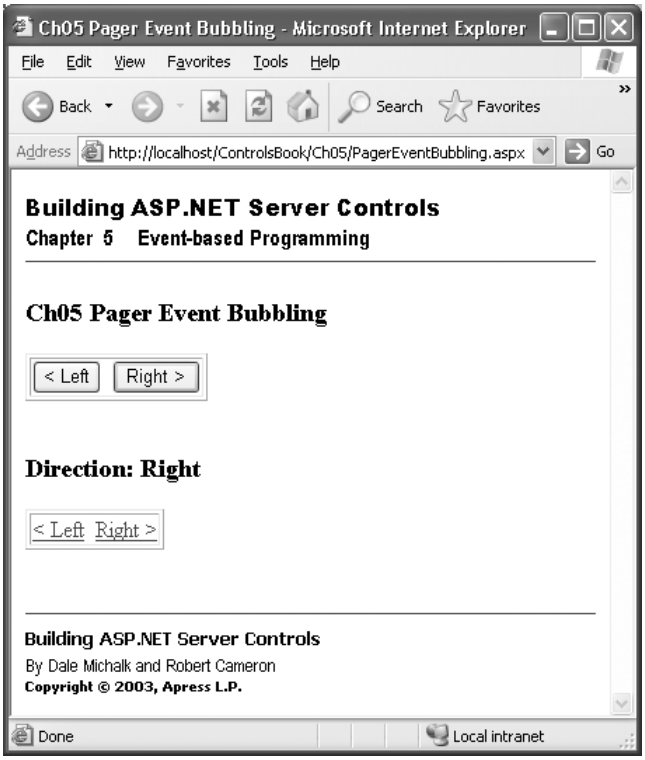

Figure 5-20. The Page Event Bubbling Web Form after clicking the Right hyperlink button

A snippet from the rendered HTML shows that the pager1 and pager2 Pager controls from the Pager Event Bubbling Web Form have their child controls identified in a nested fashion due to the INamingContainer interface with ASP.NET generating the UniqueID property:

```
<INPUT type="submit" name="pager1:buttonLeft" id="pager1:buttonLeft"
          value="&lt; Left"/>  
<INPUT type="submit" name="pager1:buttonRight" id="pager1:buttonRight"
          value="Right &gt;"
<A href="javascript:__doPostBack('pager2:buttonLeft','')">&lt; Left
</A>  <A href="javascript:__doPostBack('pager2:buttonRight','')">
Right &gt;</A></td></tr></table><br>
```

In the final section of this chapter, we review the control life cycle, which provides orderly processing to the busy life of server controls.

Control Life Cycle

The examples so far have demonstrated the use of server-side events to coordinate the activities of an ASP.NET application as part of an .aspx page. Each HTTP request/response cycle that the page executes follows a well-defined process known as the *control execution life cycle*. The `Page` server control orchestrates these activities on behalf of all the server controls in the `Page`'s control tree. Control developers need to understand the flow of execution to ensure that their custom controls perform as expected as part of an ASP.NET Web Form. Figure 5-21 provides a high-level view of the page life cycle process.

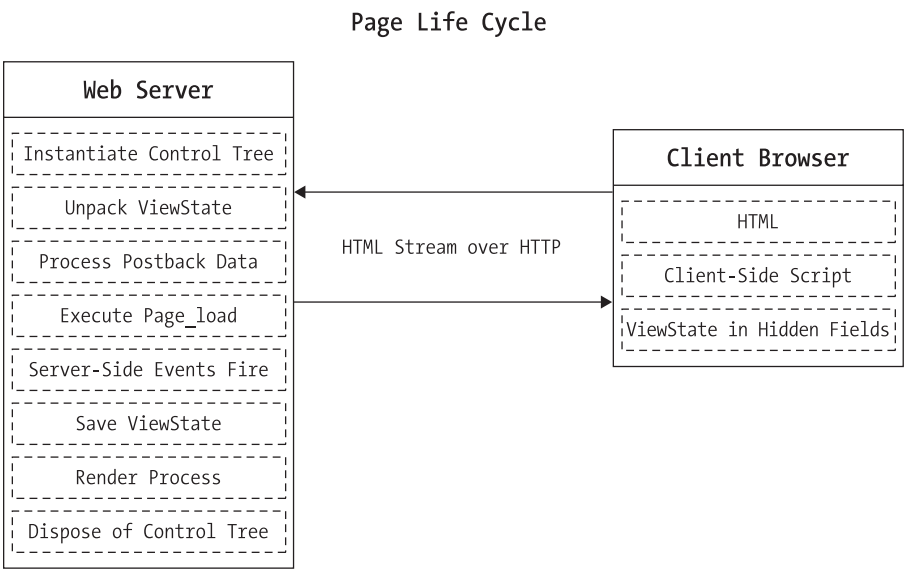

Figure 5-21. An overview of the page life cycle

After the initial page request as an HTTP GET, each subsequent HTTP POST page request/response cycle generally consists of the following steps:

1. Instantiate the control tree, creating each server control object.
2. Unpack ViewState for each server control object.
3. Set the state from the previous server-side processing cycle for each object in the tree.
4. Process postback data.
5. Handle the `Page_Load` event.
6. Let controls know that data changed through postback, updating control state as necessary.

7. Execute server-side events based on data changes from postback.
8. Persist state back to ViewState.
9. Execute the render process for each server control.
10. Unload the page and its control tree.

This process is what provides the illusion of a stateful application to the end user. During each request/response round-trip, state is unpacked, changes are processed, the UI is updated, and the page is sent back to the user's browser with its new state values embedded in a hidden form field as ViewState, ready for the next request/response cycle. We next examine what events are available to controls as the page life cycle executes on the server side.

Plugging Into the Life Cycle

Server controls have a well-defined behavior pattern that coincides with the overall page life cycle. The ASP.NET framework provides a series of events that server controls can override to customize behavior during each phase of the life cycle. Table 5-1 provides an overview of each of these events.

Table 5-1. Server Control Events Related to the Control Execution Life Cycle

SERVER CONTROL EVENT	PAGE LIFE CYCLE PHASE	DESCRIPTION
`Init`	Initialization	Initializes settings for the control.
`LoadViewState`	Unpack ViewState	Populates the state values of the control from ViewState.
`LoadPostData`	Handle form postback data	Updates control's state values from posted data.
`Load`	`Page_Load` event	Executes code common to every page request/response cycle.
`RaisePostDataChangedEvent`	Initialization for server-side events	Notifies control that newly posted data changed its state.
`RaisePostBackEvent`	Execute server-side events	Goes hand-in-hand with previous event. Server-side events fire as a result of changes found in posted data for a particular control.
`PreRender`	Render process	Allows each control a chance to update state values before rendering.
`SaveViewState`	Save ViewState	Persists a control's updated state through the ViewState mechanism.
`Render`	Render process	Generates HTML reflecting the control's state and settings.
`Dispose`	Dispose of control tree	Releases any resources held by the control before teardown.

As you can see in Table 5-1, ASP.NET provides each server control the capability to finely tune each phase in the life cycle. You can choose to accept default behavior, or you can customize a particular phase by overriding the appropriate event.

The Lifecycle Server Control

Now that we have covered the basics of the control execution life cycle, we are going to examine this process in more detail by overriding all available events in a server control named `Lifecycle`. The overridden methods fall into two camps: those that raise defined events exposed by a control and those that are not events but perform a necessary action for the control.

`OnInit`, `OnLoad`, `OnPreRender`, and `OnUnload` are events defined in `System.Web.UI.Control` that a control developer can override as required for a particular control. `LoadViewState`, `LoadPostData`, `RaisePostDataChangedEvent`, `RaisePostBackEvent`, `TrackViewState`, `SaveViewState`, and `Render` are all events that perform necessary actions for the control to maintain its state and event processing.

CAUTION *As with most object-oriented class hierarchies, it is usually (though not always) necessary to call the base class's version of an overridden method in the descendent class to ensure consistent behavior. If the base method is not called in the descendent class, instances of that class will most likely fail to behave as expected—or worse, they could cause instability.*

The implementation of `Dispose` deviates from the previous description for overridden methods. The `Control` class does expose a `Dispose` event, but it does not have an `OnDispose` method to raise it. Instead, providing a `Dispose` method follows the design pattern for objects that work with scarce resources, implementing the `IDisposable` interface.

Life Cycle and the HTTP Protocols GET and POST

The page life cycle differs based on whether the Web Form is requested for the first time via an HTTP GET or instead is initiated as part of a postback resulting from an HTTP POST generated by a control element on the page submitting the Web Form back to the server. The HTTP POST generally causes more life cycle activities because of the requirement to process data posted by the client back to the web server, raising events associated with state changes.

Figure 5-22 shows the two variants (initial GET versus POST) of the Web Form life cycle and the names of the phases we discuss in detail shortly.

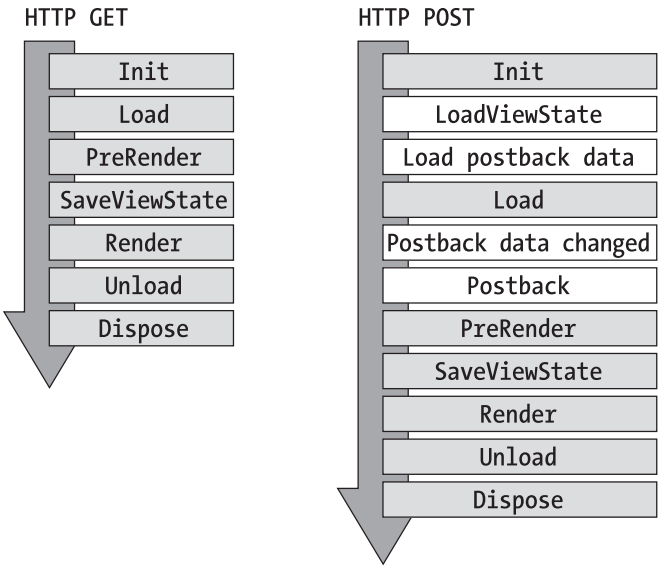

Figure 5-22. The control life cycle

In order to discuss the control life cycle, we use a control that overrides the methods necessary to track the execution of each of the life cycle events as they occur. Listing 5-19 provides the class file for the `Lifecycle` control that handles this task. The implementation of each overridden method is quite simple, with a call to the trace function notifying us that the method is executing.

Listing 5-19. The `Lifecycle` *Control Class File*

```
using System;
using System.Web.UI;
using System.Collections.Specialized;
using System.Diagnostics;

namespace ControlsBookLib.Ch05
{
   [ToolboxData("<{0}:Lifecycle runat=server></{0}:Lifecycle>")]
   public class Lifecycle : Control, IPostBackEventHandler, IPostBackDataHandler
   {
      // Init Event
      override protected void OnInit(System.EventArgs e)
      {
         Trace("Lifecycle: Init Event.");
         base.OnInit(e);
      }

      override protected void TrackViewState()
      {
         Trace("Lifecycle: Track ViewState.");
         base.TrackViewState();
```

```
}

// Load ViewState Event
override protected void LoadViewState(object savedState)
{
   Trace("Lifecycle: Load ViewState Event.");
   base.LoadViewState(savedState);
}

// Load Postback Data Event
public bool LoadPostData(string postDataKey,
   NameValueCollection postCollection)
{
   Trace("Lifecycle: Load PostBack Data Event.");

   Page.RegisterRequiresRaiseEvent(this);
   return true;
}

// Load Event
override protected void OnLoad(System.EventArgs e)
{
   Trace("Lifecycle: Load Event.");
   base.OnLoad(e);
}

// Post Data Changed Event
public void RaisePostDataChangedEvent()
{
   Trace("Lifecycle: Post Data Changed Event.");
}

// Postback Event
public void RaisePostBackEvent(string argument)
{
   Trace("Lifecycle: PostBack Event.");
}

// PreRender Event
override protected void OnPreRender(System.EventArgs e)
{
   Trace("Lifecycle: PreRender Event.");
   Page.RegisterRequiresPostBack(this);
   base.OnPreRender(e);
}

// Save ViewState
override protected object SaveViewState()
{
   Trace("Lifecycle: Save ViewState.");
   return base.SaveViewState();
}

// Render Event
override protected void Render(HtmlTextWriter writer)
{
```

```
            base.Render(writer);
            Trace("Lifecycle: Render Event.");
            writer.Write("<h3>LifeCycle Control</h3>");

        }

        // Unload Event
        override protected void OnUnload(System.EventArgs e)
        {
            Trace("Lifecycle: Unload Event.");
            base.OnUnload(e);
        }

        // Dispose Event
        public override void Dispose()
        {
            Trace("Lifecycle: Dispose Event.");
            base.Dispose();
        }

        private void Trace(string info)
        {
            Context.Trace.Warn(info);
            Debug.WriteLine(info);
        }
    }
}
```

Listings 5-20 and 5-21 outline the Web Form that hosts the control, with the ASP.NET tracing mechanism turned on. The UI appearance is a single button on the Web Form with trace output turned on.

Listing 5-20. The Life Cycle Web Form .aspx Page File

```
<%@ Register TagPrefix="apressUC" TagName="ControlsBookFooter"
Src="..\ControlsBookFooter.ascx" %>
<%@ Register TagPrefix="apressUC" TagName="ControlsBookHeader"
Src="..\ControlsBookHeader.ascx" %>
<%@ Register TagPrefix="apress" Namespace="ControlsBookLib.Ch05"
Assembly="ControlsBookLib" %>
<%@ Page Trace="true" language="c#" Codebehind="LifeCycle.aspx.cs"
        AutoEventWireup="false" Inherits="ControlsBookWeb.Ch05.LifeCycle" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>Ch05 Lifecycle</title>
    <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
         content="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>
  <body MS_POSITIONING="FlowLayout">
    <form id="LifeCycle" method="post" runat="server">
```

```
      <apressUC:ControlsBookHeader id="Header" runat="server" ChapterNumber="5"
                      ChapterTitle="Event-based Programming" />
      <h3>Ch05 Lifecycle</h3>
      <apress:Lifecycle id="life1" runat="server" />
      <asp:Button id="Button1" runat="server" Text="Button"></asp:Button>
      <apressUC:ControlsBookFooter id="Footer" runat="server" />
    </form>
  </body>
</HTML>
```

Listing 5-21. The Life Cycle Web Form Code-Behind Class File

```
using System;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace ControlsBookWeb.Ch05
{

   public class LifeCycle : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.Button Button1;
      protected ControlsBookLib.Ch05.Lifecycle life1;

      private void Page_Load(object sender, System.EventArgs e)
      {

      }

      #region Web Form Designer generated code
      override protected void OnInit(EventArgs e)
      {
         //
         // CODEGEN: This call is required by the ASP.NET Web Form Designer.
         //
         InitializeComponent();
         base.OnInit(e);
      }

      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.Load += new System.EventHandler(this.Page_Load);

      }
      #endregion
   }
}
```

The first execution of the Life Cycle Web Form results in an HTTP GET protocol request and generates the life cycle events shown in the ASP.NET Trace output of Figure 5-23.

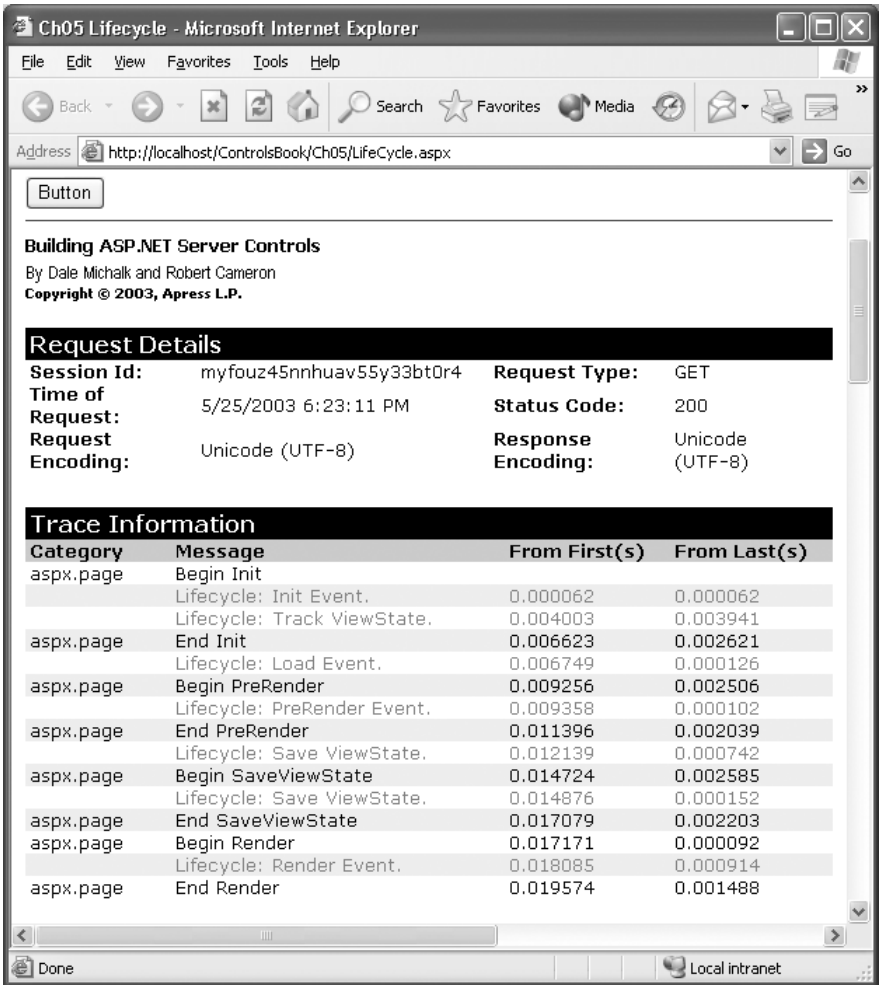

Figure 5-23. The Lifecycle.aspx trace output from an HTTP GET request

We next cover the life cycle events that occur when an HTTP GET request occurs, starting with the `Init` event.

Init Event

The first phase processed by the control is the `Init` event. We are notified of this phase by overriding the protected `OnInit` method inherited from the base class `System.Web.UI.Control`:

```
override protected void OnInit(System.EventArgs e)
{
   base.OnInit();
   Trace("Lifecycle:  Init Event.");
}
```

The code in the `OnInit` method uses a private utility method called `Trace` that sends status information to the `Trace` class via the control's `Context` property available to ASP.NET server controls:

```
private void Trace(string info)
{
   Page.Trace.Warn(info);
   Debug.WriteLine(info);
}
```

This class method also sends output to the debug stream via the `System.Diagnostics.Debug` class and its `WriteLine` method. The reason for this extra step is to view `Unload` and `Dispose` event execution, which occurs after the Web Form is finished writing out its content via the `Render` method and the ASP.NET trace tables have been generated. You can view debug stream information in the Output window of Visual Studio .NET when debugging.

The `Init` event is an opportunity for the control to initialize any resources it needs to service the page request. A control can access any child controls in this method if necessary; however, peer- and parent-level controls are not guaranteed to be accessible at this point in the life cycle.

Overriding methods such as `OnInit` from the base class `System.Web.UI.Control` requires that we call the base version of this method to ensure proper functioning of the event. The base class implementation of `OnInit` actually raises the `Init` event exposed by the root `Control` class to clients.

If you override `OnInit` but do not call the base class version of this event, the event will not be raised to clients that are registered to receive it. This applies to the other `On`-prefixed methods such as `OnLoad`, `OnPreRender`, and `OnUnload`, which are part of the life cycle process, as well as other non-life-cycle-specific event methods such as `OnDataBinding` and `OnBubbleEvent`.

TrackViewState Method

The TrackViewState method executes immediately after initialization and marks the beginning of ViewState processing, and state tracking, in the control life cycle. If you have attribute values that you do not want to save in ViewState across page round-trips for efficiency purposes, you should set these values in the OnInit method. Otherwise, all control property value modifications performed after this method executes will persist to ViewState.

If desired, you can make modifications to state values in this method that won't be marked as dirty as long as you do so before executing the inherited base.TrackViewState method or before calling the encapsulated StateBag.TrackViewState method.

Load Event

The Load event should be quite familiar to you because we have leveraged this convenient location for common page code in our Web Forms in previous examples. It is a handy place to put page initialization logic because you are guaranteed that all controls in the Page's control tree are created and all state-loading mechanisms have restored each control's state back to where it was at the end of the previous request/response page round-trip. This event also occurs before any controls in the Page's control tree fire their specific events resulting from value changes in postback data. To customize control behavior in this phase, override the OnLoad method.

PreRender Event

The PreRender event is a phase in the control life cycle that represents the last-ditch chance for a control to do something before it is rendered. This is the location to put code that must execute before rendering but after the Load event, state management methods, and postback events have occurred. Controls can override the OnPreRender method for this special situation. Note that changes made to a control's state at this point in the life cycle will persist to ViewState.

SaveViewState Method

The SaveViewState method saves the ViewState dictionary by default without any additional action by you. Overriding this method is only necessary when a control needs to customize state persistence in ViewState. This method is called only when the EnableViewState property inherited from Control is set to true. The object instance that is returned from this method is serialized by ASP.NET into the final ViewState string that is emitted into the page's __VIEWSTATE hidden field. Be aware that SaveViewState is called twice in our sample code as a result of enabling page tracing, which makes a call to SaveViewState to gather information for tracing purposes. With tracing disabled during normal page execution, SaveViewState is called only once.

Render Method

You are by now very familiar with overriding the `Render` method in a custom control to generate a control's UI. The `HtmlTextWriter` class does the bulk of the work here, writing out the control as HTML and script where applicable to the HTTP response stream. Note that any changes to a control's state made within this method will render into the UI but will not be saved as part of ViewState.

Unload Event

The `Page` class implements this method to perform cleanup. Overriding the `OnUnload` method from the base control class allows the control to hook into this event. Although the `Unload` event is an opportunity for a control to release any resources that it has obtained in earlier control events such as `Init` or `Load`, it is recommended that you release resources in its `Dispose` method.

The trace output from the ASP.NET page does not display any information pertaining to this event because it fires after `Render` executes, but we can use the debug stream output from the Output window when debugging the Web Form in Visual Studio .NET to see the result:

```
Lifecycle: Init Event.
Lifecycle: Track ViewState.
Lifecycle: Load Event.
Lifecycle: PreRender Event.
Lifecycle: Save ViewState.
Lifecycle: Save ViewState.
Lifecycle: Render Event.
Lifecycle: Unload Event.
Lifecycle: Dispose Event.
```

Dispose Method

`Dispose` is the recommended location for cleaning up resources. Implementing a `Dispose` method is recommended in .NET Framework programming when unmanaged resources (such as a connection to SQL Server) are acquired by a control and need to be safely released within the garbage collection architecture. The pattern is based on the `IDispose` interface that gives a way for clients to tell an object to clean up its unmanaged resources:

```
Interface IDispose
{
   void Dispose();
}
```

Once a client is finished working with an object, the client notifies the object it is finished by calling the object's `Dispose` method. This gives the object immediate confirmation that it can clean up its resources instead of waiting for its `Finalize` method to be called during garbage collection. Because `Dispose` is the design pattern common in .NET, it is recommended that you implement cleanup in `Dispose` instead of `Unload` to release unmanaged resources.

HTTP POST Request via Postback

Additional events and methods of the control life cycle are exercised once we execute a postback of the Life Cycle Web Form by clicking the button. The output of the trace is much larger, so the screen shot in Figure 5-24 is filled by that table.

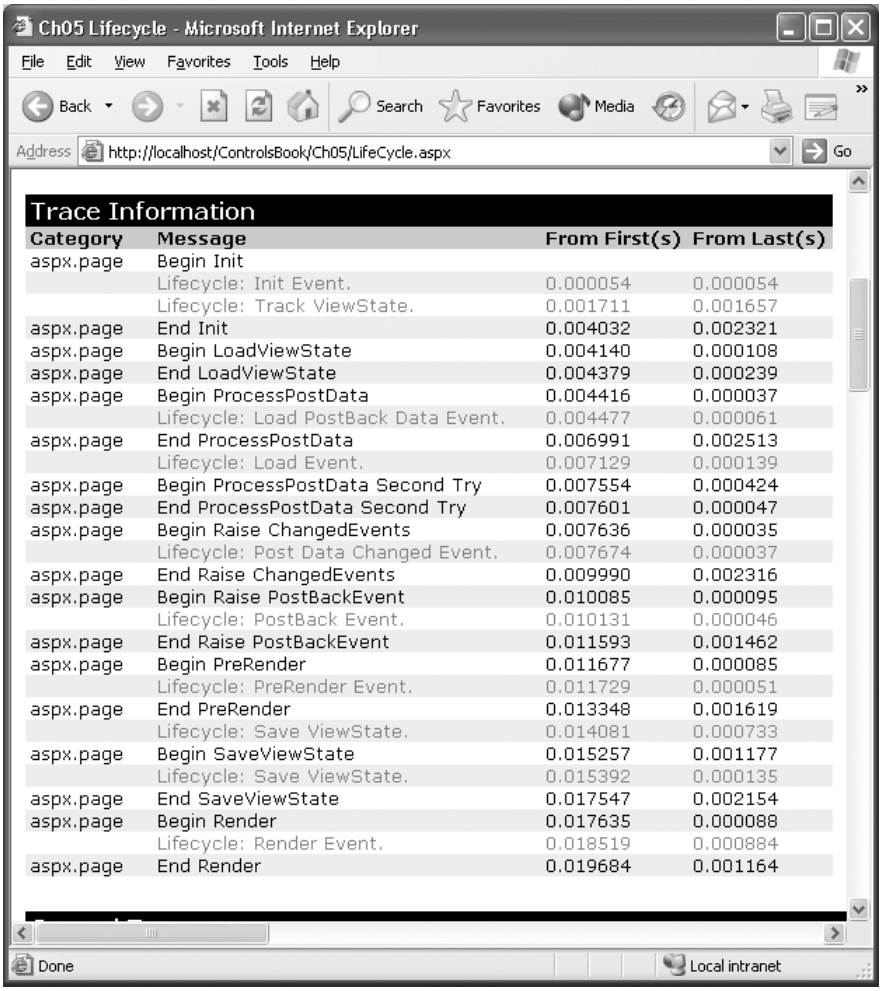

Figure 5-24. The Lifecycle.aspx Trace output from an HTTP POST postback

The output from the Visual Studio .NET Debug window confirms the sequence of events as well:

```
Lifecycle: Init Event.
Lifecycle: Track ViewState.
Lifecycle: Load PostBack Data Event.
Lifecycle: Load Event.
Lifecycle: Post Data Changed Event.
Lifecycle: PostBack Event.
Lifecycle: PreRender Event.
Lifecycle: Save ViewState.
Lifecycle: Save ViewState.
Lifecycle: Render Event.
Lifecycle: Unload Event.
Lifecycle: Dispose Event.
```

LoadViewState Method

Overriding the `LoadViewState` method is necessary if a control has previously overridden `SaveViewState` to customize ViewState serialization. Customization of the ViewState persistence mechanism is commonly performed by developers in more complex controls that have complex properties such as a reference type or a collection of objects. The decision to customize ViewState really comes down to whether or not a control's state can be easily or efficiently reduced to a string representation.

LoadPostBackData Method

In the previous chapter, we discussed how to retrieve client form post data via implementation of the `IPostBackDataHandler` interface. The `LoadPostData` routine is given the opportunity to process the postback data and to update the control's state. It also allows the control to notify ASP.NET that it wishes to raise an event at a later time in order to permit clients a chance to process the state change. For our purposes, the `Lifecycle` control always returns true, so the change event is always raised.

Keen observers will notice that we really should not be receiving the form post information, because we did not emit an HTML tag such as `<INPUT>`. However, we greased the wheels in the ASP.NET framework by calling `Page.RegisterRequiresPostBack` in `OnPreRender`:

```
override protected void OnPreRender(System.EventArgs e)
{
   base.OnPreRender(e);
   Trace("Lifecycle: PreRender Event.");
   Page.RegisterRequiresPostBack(this);
}
```

This makes it possible to receive a call to our LoadPostData method by ASP.NET. We perform a similar task in LoadPostData to ensure we receive the PostBack event by calling Page.RegisterRequiresRaiseEvent:

```
public bool LoadPostData(string postDataKey, NameValueCollection postCollection)
{
   Trace("Lifecycle: Load PostBack Data Event.");
   Page.RegisterRequiresRaiseEvent(this);
   return true;
}
```

RaisePostDataChangedEvent Method

For controls that have state changes reflected in postback data, these controls most likely need to raise server-side events. These events are raised from within the RaisePostDataChangedEvent method of each respective control. Following this design guideline ensures that control state is restored from ViewState and updated from postback data before the various events begin to fire. Raising server-side events from within any other control method can cause hard-to-debug side effects for event consumers. This routine is called only if the LoadPostData returns true.

RaisePostBackEvent Method

Implementing RaisePostBackEvent ensures that the server control notifies ASP.NET that the state of the control has changed. To participate in postback processing, a control must implement the IPostBackEventHandler interface. Controls implement this interface and emit some sort of HTML to submit the Web Form back to the server, whether via a button or an HTML element with JavaScript code to submit the form programmatically.

In our sample Lifecycle control, we rigged the system by calling Page.RegisterRequiresRaiseEvent in the LoadPostData method. Our SuperButton control sample in this chapter demonstrates how to execute this properly. We use this shortcut to hook into this event for purposes discussing the control life cycle.

Summary

In this chapter, we discussed how to implement events in the ASP.NET framework. Event-based programming is a critical aspect of ASP.NET development, making web development more like developing with Visual Basic on the Windows desktop. We also discussed how to bubble events up the control hierarchy and we explored the control life cycle.

System.EventHandler is the default delegate type for events in ASP.NET. Inherit from this type when you create custom event handlers so that your controls behave in a similar manner to the built-in controls.

Events are generally invoked in a control through a `virtual protected` method that prefixes the word "On" to the event name to create a method name such as `OnClick` and `OnTextChanged`. Custom events implement their own delegate type with the name suffixed by "EventHandler". Custom events can also implement a custom `EventArgs`-derived class to provide event data tailored to the particular situation. The simplest way for a control to expose an event is to declare one as a public field of a custom control class.

Controls can use the `Events` collection inherited from `System.Web.UI.Control` to efficiently manage events in a sparse collection instead of the one-field-per-event model. Using the `Events` collection along with custom event registration code can potentially save a large amount of memory for a control with many events that are not all normally implemented.

`Command` events are a special event type used by list controls in ASP.NET to simplify handling buttons as child controls. `Command` events expose `CommandName` and `CommandArgument` properties to communicate their intentions to the parent control.

Event bubbling is a concept in ASP.NET whereby a control can raise an event through its parent control hierarchy. `RaiseBubbleEvent` starts the event in motion. Parent controls can catch the event by overriding `OnBubbleEvent`. `RaisePostBackEvent` is the method in `IPostBackEventHandler` that allows a control to capture a postback generated by a change in data.

`INamingContainer` is used by a composite control to ensure that its child controls have a unique name on the page even if the composite control is used several times on the page via the `UniqueID` property.

Controls follow a well-defined life cycle execution process to help coordinate events and activities. Understanding the control life cycle will ensure your custom controls behave as expected. The complete control life cycle for an HTTP GET request includes these events in order: `Init`, `TrackViewState`, `Load`, `PreRender`, `SaveViewState`, `Render`, `Unload`, and `Dispose`. The complete control life cycle for an HTTP POST request includes these events in order: `Init`, `TrackViewState`, `LoadViewState`, `LoadPostData`, `Load`, `RaisePostDataChangedEvent`, `PostBack`, `PreRender`, `SaveViewState`, `Render`, `Unload`, and `Dispose`.