1

Application and Page Frameworks

COPYRIGHTED MATERIAL

The evolution of ASP.NET continues! The progression from Active Server Pages 3.0 to ASP.NET 1.0 was revolutionary, to say the least. And now the revolution continues with the latest release of ASP.NET — version 3.5. The original introduction of ASP.NET 1.0 fundamentally changed the Web programming model. ASP.NET 3.5 is just as revolutionary in the way it will increase your productivity. As of late, the primary goal of ASP.NET is to enable you to build powerful, secure, dynamic applications using the least possible amount of code. Although this book covers the new features provided by ASP.NET 3.5, it also covers all the offerings of ASP.NET technology.

If you are new to ASP.NET and building your first set of applications in ASP.NET 3.5, you may be amazed by the vast amount of wonderful server controls it provides. You may marvel at how it enables you to work with data more effectively using a series of data providers. You may be impressed at how easily you can build in security and personalization.

The outstanding capabilities of ASP.NET 3.5 do not end there, however. This chapter looks at many exciting options that facilitate working with ASP.NET pages and applications. One of the first steps you, the developer, should take when starting a project is to become familiar with the foundation you are building on and the options available for customizing that foundation.

Application Location Options

With ASP.NET 3.5, you have the option — using Visual Studio 2008 — to create an application with a virtual directory mapped to IIS or a standalone application outside the confines of IIS. Whereas the early Visual Studio .NET 2002/2003 IDEs forced developers to use IIS for all Web applications, Visual Studio 2008 (and Visual Web Developer 2008 Express Edition, for that matter) includes a built-in Web server that you can use for development, much like the one used in the past with the ASP.NET Web Matrix.

This built-in Web server was previously presented to developers as a code sample called Cassini. In fact, the code for this mini Web server is freely downloadable from the ASP.NET team Web site found at `www.asp.net`.

The following section shows you how to use the built-in Web server that comes with Visual Studio 2008.

Built-In Web Server

By default, Visual Studio 2008 builds applications without the use of IIS. You can see this when you select New ➪ Web Site in the IDE. By default, the location provided for your application is in `C:\Users\Bill\Documents\Visual Studio 2008\WebSites` if you are using Windows Vista (shown in Figure 1-1). It is not `C:\Inetpub\wwwroot\` as it would have been in Visual Studio .NET 2002/2003. By default, any site that you build and host inside `C:\Users\Bill\Documents\Visual Studio 2008\WebSites` (or any other folder you create) uses the built-in Web server that is part of Visual Studio 2008. If you use the built-in Web server from Visual Studio 2008, you are not locked into the `WebSites` folder; you can create any folder you want in your system.

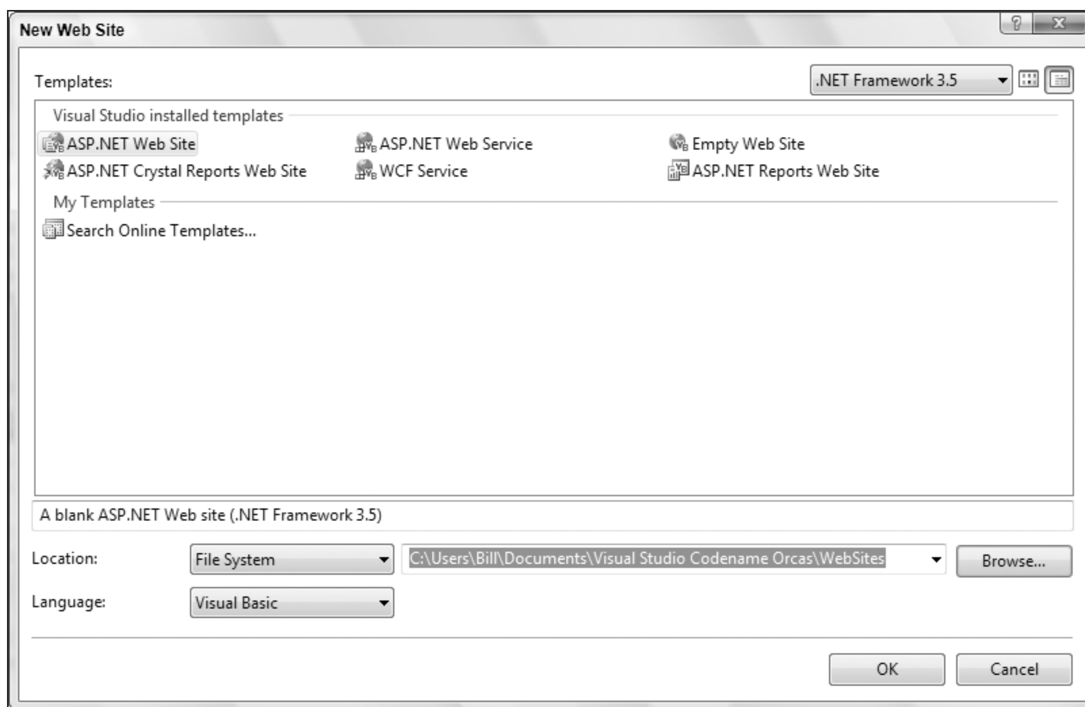

Figure 1-1

To change from this default, you have a handful of options. Click the Browse button in the New Web Site dialog. This brings up the Choose Location dialog, shown in Figure 1-2.

If you continue to use the built-in Web server that Visual Studio 2008 provides, you can choose a new location for your Web application from this dialog. To choose a new location, select a new folder and save

your .aspx pages and any other associated files to this directory. When using Visual Studio 2008, you can run your application completely from this location. This way of working with the ASP.NET pages you create is ideal if you do not have access to a Web server because it enables you to build applications that do not reside on a machine with IIS. This means that you can even develop ASP.NET applications on operating systems such as Windows XP Home Edition.

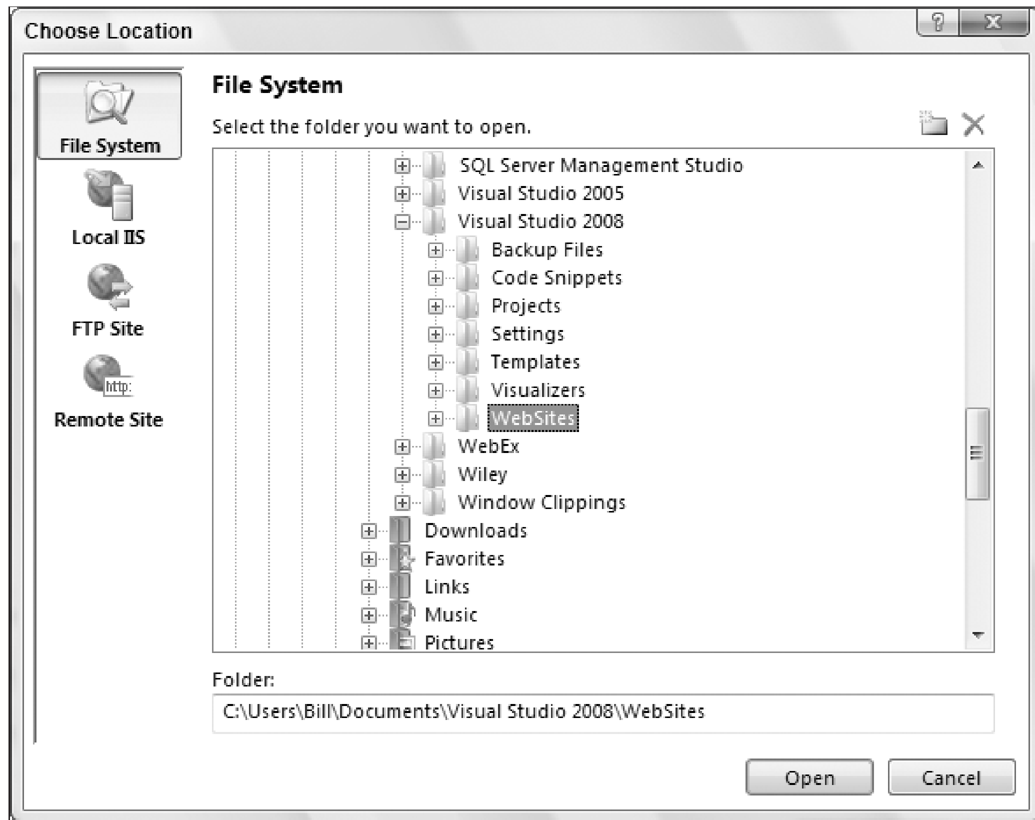

Figure 1-2

IIS

From the Choose Location dialog, you can also change where your application is saved and which type of Web server your application employs. To use IIS (as you probably did when you used Visual Studio .NET 2002/2003), select the Local IIS button in the dialog. This changes the results in the text area to show you a list of all the virtual application roots on your machine. You are required to run Visual Studio as an administrator user if you want to see your local IIS instance.

To create a new virtual root for your application, highlight Default Web Site. Two accessible buttons appear at the top of the dialog box (see Figure 1-3). When you look from left to right, the first button in the upper-right corner of the dialog box is for creating a new Web application — or a virtual root. This button is shown as a globe inside a box. The second button enables you to create virtual directories for

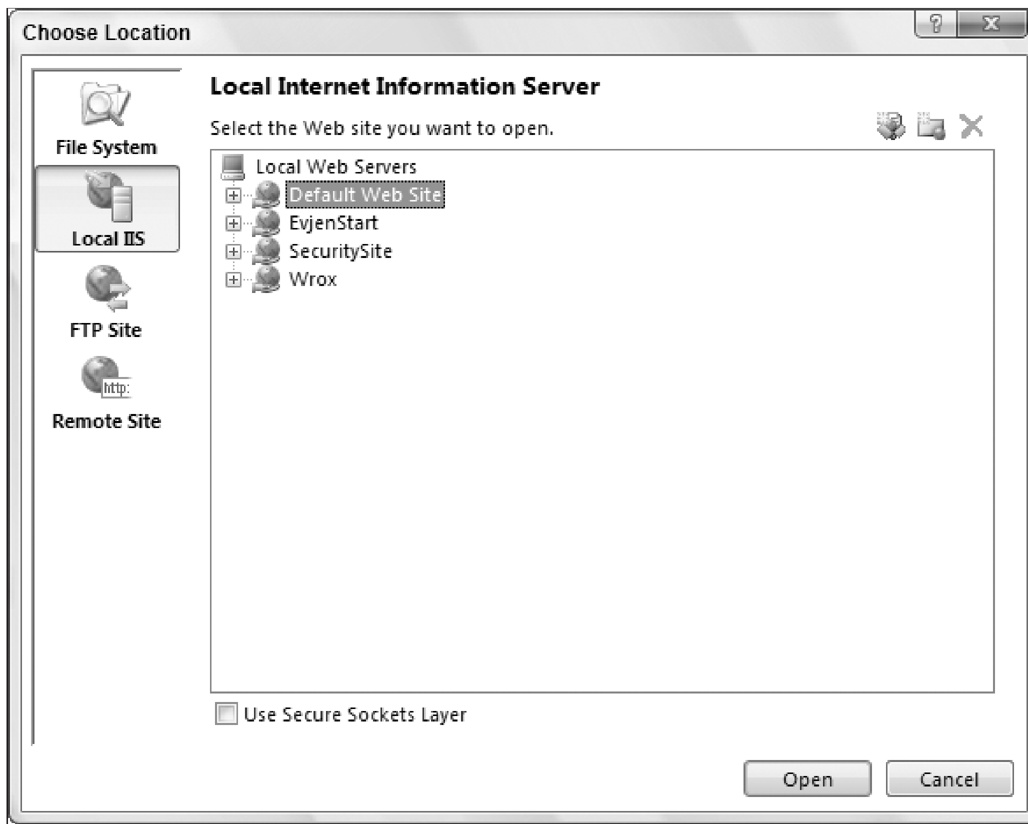

Figure 1-3

any of the virtual roots you created. The third button is a Delete button, which allows you to delete any selected virtual directories or virtual roots on the server.

After you have created the virtual directory you want, click the Open button. Visual Studio 2008 then goes through the standard process to create your application. Now, however, instead of depending on the built-in Web server from ASP.NET 3.5, your application will use IIS. When you invoke your application, the URL now consists of something like `http://localhost/MyWeb/Default.aspx`, which means it is using IIS.

FTP

Not only can you decide on the type of Web server for your Web application when you create it using the Choose Location dialog, but you can also decide where your application is going to be located. With the previous options, you built applications that resided on your local server. The FTP option enables you to actually store and even code your applications while they reside on a server somewhere else in your enterprise — or on the other side of the planet. You can also use the FTP capabilities to work on different locations within the same server. Using this new capability provides a wide range of possible options.

The built-in capability giving FTP access to your applications is a major enhancement to the IDE. Although formerly difficult to accomplish, this task is now quite simple, as you can see from Figure 1-4.

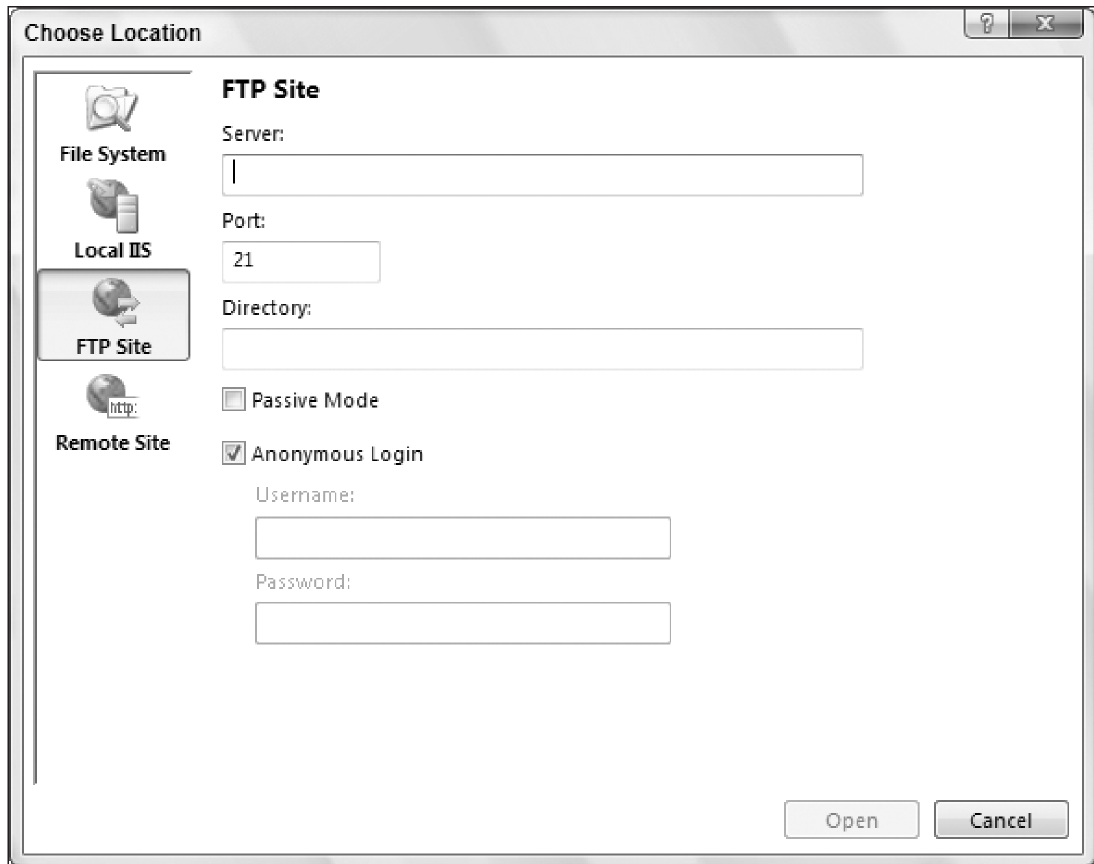

Figure 1-4

To create your application on a remote server using FTP, simply provide the server name, the port to use, and the directory — as well as any required credentials. If the correct information is provided, Visual Studio 2008 reaches out to the remote server and creates the appropriate files for the start of your application, just as if it were doing the job locally. From this point on, you can open your project and connect to the remote server using FTP.

Web Site Requiring FrontPage Extensions

The last option in the Choose Location dialog is the Remote Sites option. Clicking this button provides a dialog that enables you to connect to a remote or local server that utilizes FrontPage Extensions. This option is displayed in Figure 1-5.

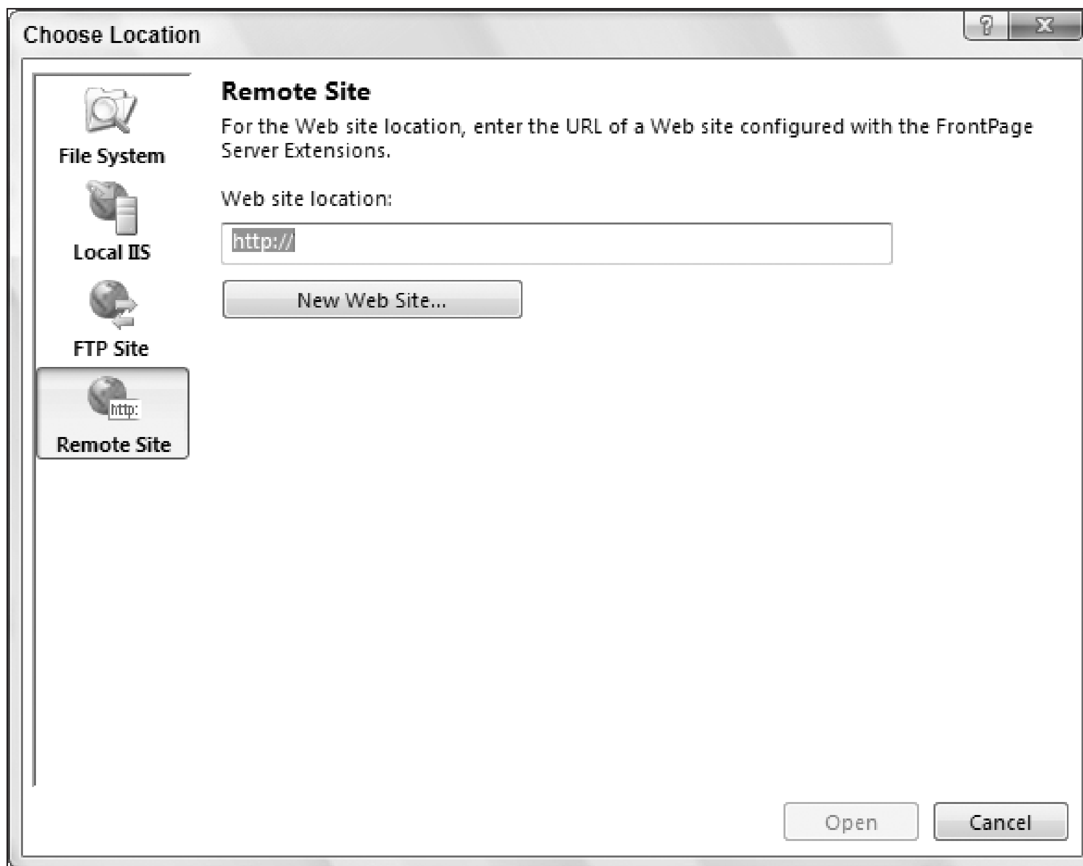

Figure 1-5

The ASP.NET Page Structure Options

ASP.NET 3.5 provides two paths for structuring the code of your ASP.NET pages. The first path utilizes the code-inline model. This model should be familiar to classic ASP 2.0/3.0 developers because all the code is contained within a single `.aspx` page. The second path uses ASP.NET's code-behind model, which allows for code separation of the page's business logic from its presentation logic. In this model, the presentation logic for the page is stored in an `.aspx` page, whereas the logic piece is stored in a separate class file: `.aspx.vb` or `.aspx.cs`. It is considered best practice to use the code-behind model as it provides a clean model in separation of pure UI elements from code that manipulates these elements. It is also seen as a better means in maintaining code.

One of the major complaints about Visual Studio .NET 2002 and 2003 is that it forced you to use the code-behind model when developing your ASP.NET pages because it did not understand the code-inline model. The code-behind model in ASP.NET was introduced as a new way to separate the presentation code and business logic. Listing 1-1 shows a typical `.aspx` page generated using Visual Studio .NET 2002 or 2003.

Listing 1-1: A typical .aspx page from ASP.NET 1.0/1.1

```
<%@ Page Language="vb" AutoEventWireup="false" Codebehind="WebForm1.aspx.vb"
    Inherits="WebApplication.WebForm1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
 <HEAD>
    <title>WebForm1</title>
    <meta name="GENERATOR" content="Microsoft Visual Studio .NET 7.1">
    <meta name="CODE_LANGUAGE" content="Visual Basic .NET 7.1">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
 </HEAD>
 <body>
    <form id="Form1" method="post" runat="server">
       <P>What is your name?<br>
       <asp:TextBox id="TextBox1" runat="server"></asp:TextBox><BR>
       <asp:Button id="Button1" runat="server" Text="Submit"></asp:Button></P>
       <P><asp:Label id="Label1" runat="server"></asp:Label></P>
    </form>
 </body>
</HTML>
```

The code-behind file created within Visual Studio .NET 2002/2003 for the `.aspx` page is shown in Listing 1-2.

Listing 1-2: A typical .aspx.vb/.aspx.cs page from ASP.NET 1.0/1.1

```
Public Class WebForm1
    Inherits System.Web.UI.Page

#Region " Web Form Designer Generated Code "

    'This call is required by the Web Form Designer.
    <System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()

    End Sub
    Protected WithEvents TextBox1 As System.Web.UI.WebControls.TextBox
    Protected WithEvents Button1 As System.Web.UI.WebControls.Button
    Protected WithEvents Label1 As System.Web.UI.WebControls.Label

    'NOTE: The following placeholder declaration is required by the Web Form
        Designer.
    'Do not delete or move it.
    Private designerPlaceholderDeclaration As System.Object

    Private Sub Page_Init(ByVal sender As System.Object, ByVal e As
      System.EventArgs) Handles MyBase.Init
        'CODEGEN: This method call is required by the Web Form Designer
        'Do not modify it using the code editor.
        InitializeComponent()
    End Sub
```

Continued

```
#End Region

    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
      System.EventArgs) Handles MyBase.Load
        'Put user code to initialize the page here
    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
      System.EventArgs) Handles Button1.Click
        Label1.Text = "Hello " & TextBox1.Text
    End Sub
End Class
```

In this code-behind page from ASP.NET 1.0/1.1, you can see that a lot of the code that developers never have to deal with is hidden in the `#Region` section of the page. Because ASP.NET 3.5 is built on top of .NET 3.5, which in turn is utilizing the core .NET 2.0 Framework, it can take advantage of the .NET Framework capability of partial classes. Partial classes enable you to separate your classes into multiple class files, which are then combined into a single class when the application is compiled. Because ASP.NET 3.5 combines all this page code for you behind the scenes when the application is compiled, the code-behind files you work with in ASP.NET 3.5 are simpler in appearance and the model is easier to use. You are presented with only the pieces of the class that you need. Next, we will look at both the inline and code-behind models from ASP.NET 3.5.

Inline Coding

With the .NET Framework 1.0/1.1, developers went out of their way (and outside Visual Studio .NET) to build their ASP.NET pages inline and avoid the code-behind model that was so heavily promoted by Microsoft and others. Visual Studio 2008 (as well as Visual Web Developer 2008 Express Edition) allows you to build your pages easily using this coding style. To build an ASP.NET page inline instead of using the code-behind model, you simply select the page type from the Add New Item dialog and make sure that the Place Code in Separate File check box is unchecked. You can get at this dialog by right clicking the project or the solution in the Solution Explorer and selecting Add New Item (see Figure 1-6).

From here, you can see the check box you need to unselect if you want to build your ASP.NET pages inline. In fact, many page types have options for both inline and code-behind styles. The following table shows your inline options when selecting files from this dialog.

File Options Using Inline Coding	File Created
Web Form	`.aspx` file
AJAX Web Form	`.aspx` file
Master Page	`.master` file
AJAX Master Page	`.master` file
Web User Control	`.ascx` file
Web Service	`.asmx` file

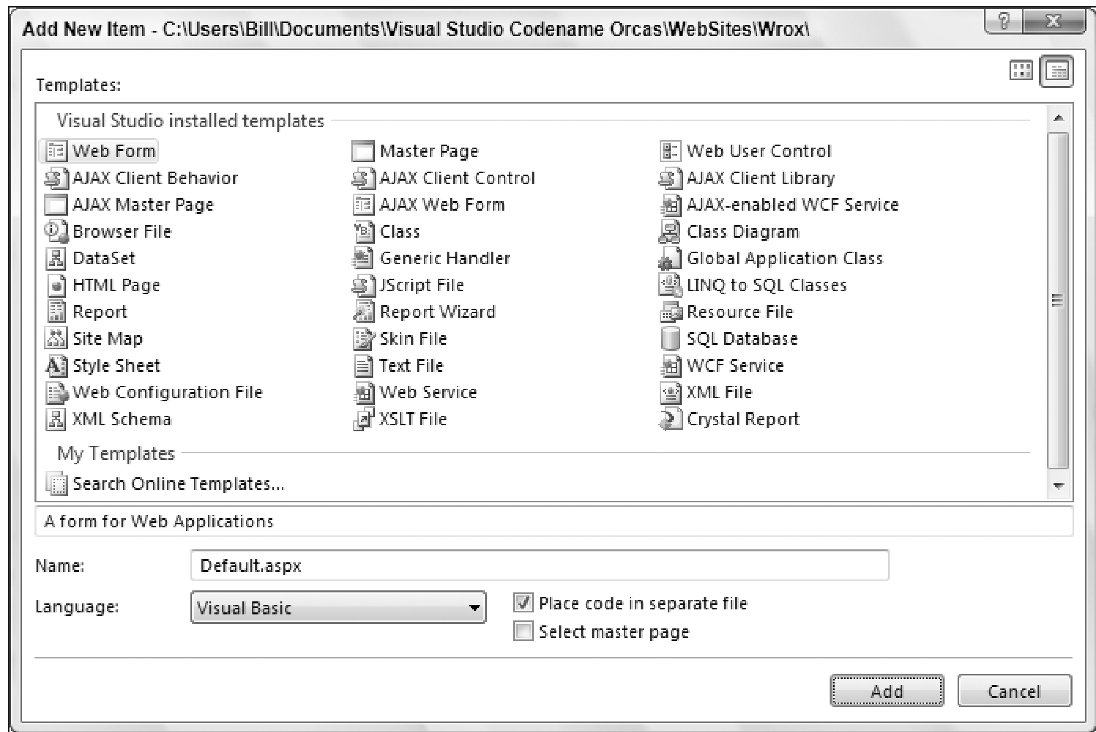

Figure 1-6

By using the Web Form option with a few controls, you get a page that encapsulates not only the presentation logic, but the business logic as well. This is illustrated in Listing 1-3.

Listing 1-3: A simple page that uses the inline coding model

VB

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    Protected Sub Button1_Click(ByVal sender As Object, _
       ByVal e As System.EventArgs)

        Label1.Text = "Hello " & Textbox1.Text
    End Sub
</script>

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

Continued

```
<body>
    <form id="form1" runat="server">
        What is your name?<br />
        <asp:Textbox ID="Textbox1" Runat="server"></asp:Textbox><br />
        <asp:Button ID="Button1" Runat="server" Text="Submit"
         OnClick="Button1_Click" />
        <p><asp:Label ID="Label1" Runat="server"></asp:Label></p>
    </form>
</body>
</html>
```

C#

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    protected void Button1_Click(object sender, System.EventArgs e)
    {
        Label1.Text = "Hello " + Textbox1.Text;
    }
</script>
```

From this example, you can see that all the business logic is encapsulated in between `<script>` tags. The nice feature of the inline model is that the business logic and the presentation logic are contained within the same file. Some developers find that having everything in a single viewable instance makes working with the ASP.NET page easier. Another great thing is that Visual Studio 2008 provides IntelliSense when working with the inline coding model and ASP.NET 3.5. Before Visual Studio 2005, this capability did not exist. Visual Studio .NET 2002/2003 forced you to use the code-behind model and, even if you rigged it so your pages were using the inline model, you lost all IntelliSense capabilities.

Code-Behind Model

The other option for constructing your ASP.NET 3.5 pages is to build your files using the code-behind model.

> *It is important to note that the more preferred method is the code-behind model rather than the inline model. This method employs the proper segmentation between presentation and business logic in many cases. You will find that many of the examples in this book use an inline coding model because it works well in showing an example in one listing. Even though the example is using an inline coding style, it is my recommendation that you move the code to employ the code-behind model.*

To create a new page in your ASP.NET solution that uses the code-behind model, select the page type you want from the New File dialog. To build a page that uses the code-behind model, you first select the page in the Add New Item dialog and make sure the Place Code in Separate File check box is checked. The following table shows you the options for pages that use the code-behind model.

File Options Using Code-Behind	File Created
Web Form	`.aspx` file `.aspx.vb` or `.aspx.cs` file
AJAX Web Form	`.aspx` file `.aspx.vb` or `.aspx.cs` file
Master Page	`.master` file `.master.vb` or `.master.cs` file
AJAX Master Page	`.master.vb` or `.master.cs` file
Web User Control	`.ascx` file `.ascx.vb` or `.ascx.cs` file
Web Service	`.asmx` file `.vb` or `.cs` file

The idea of using the code-behind model is to separate the business logic and presentation logic into separate files. Doing this makes it easier to work with your pages, especially if you are working in a team environment where visual designers work on the UI of the page and coders work on the business logic that sits behind the presentation pieces. In the earlier Listings 1-1 and 1-2, you saw how pages using the code-behind model in ASP.NET 1.0/1.1 were constructed. To see the difference in ASP.NET 3.5, look at how its code-behind pages are constructed. This is illustrated in Listing 1-4 for the presentation piece and Listing 1-5 for the code-behind piece.

Listing 1-4: An .aspx page that uses the ASP.NET 3.5 code-behind model

VB

```
<%@ Page Language="VB" AutoEventWireup="false" CodeFile="Default.aspx.vb"
    Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Simple Page</title>
</head>
<body>
    <form id="form1" runat="server">
        What is your name?<br />
        <asp:Textbox ID="Textbox1" Runat="server"></asp:Textbox><br />
        <asp:Button ID="Button1" Runat="server" Text="Submit"
         OnClick="Button1_Click" />
        <p><asp:Label ID="Label1" Runat="server"></asp:Label></p>
    </form>
</body>
</html>
```

C#

```
<%@ Page Language="C#" CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

Listing 1-5: A code-behind page

VB

```
Partial Class _Default
    Inherits System.Web.UI.Page

    Protected Sub Button1_Click(ByVal sender As Object, _
       ByVal e As System.EventArgs) Handles Button1.Click

        Label1.Text = "Hello " & TextBox1.Text
    End Sub
End Class
```

C#

```
using System;
using System.Data;
using System.Configuration;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;

public partial class _Default : System.Web.UI.Page
{
    protected void Button1_Click(object sender, EventArgs e)
    {
       Label1.Text = "Hello " + Textbox1.Text;
    }
}
```

The `.aspx` page using this ASP.NET 3.5 code-behind model has some attributes in the `Page` directive that you should pay attention to when working in this mode. The first is the `CodeFile` attribute. This is an attribute in the `Page` directive and is meant to point to the code-behind page that is used with this presentation page. In this case, the value assigned is `Default.aspx.vb` or `Default.aspx.cs`. The second attribute needed is the `Inherits` attribute. This attribute was available in previous versions of ASP.NET, but was little used before ASP.NET 2.0. This attribute specifies the name of the class that is bound to the page when the page is compiled. The directives are simple enough in ASP.NET 3.5. Look at the code-behind page from Listing 1-5.

The code-behind page is rather simple in appearance because of the partial class capabilities that .NET 3.5 provides. You can see that the class created in the code-behind file uses partial classes, employing the `Partial` keyword in Visual Basic 2008 and the `partial` keyword from C# 2008. This enables you to simply place the methods that you need in your page class. In this case, you have a button-click event and nothing else.

Later in this chapter, we look at the compilation process for both of these models.

ASP.NET 3.5 Page Directives

ASP.NET directives are something that is a part of every ASP.NET page. You can control the behavior of your ASP.NET pages by using these directives. Here is an example of the `Page` directive:

```
<%@ Page Language="VB" AutoEventWireup="false" CodeFile="Default.aspx.vb"
    Inherits="_Default" %>
```

Eleven directives are at your disposal in your ASP.NET pages or user controls. You use these directives in your applications whether the page uses the code-behind model or the inline coding model.

Basically, these directives are commands that the compiler uses when the page is compiled. Directives are simple to incorporate into your pages. A directive is written in the following format:

```
<%@ [Directive] [Attribute=Value] %>
```

From this, you can see that a directive is opened with a `<%@` and closed with a `%>`. It is best to put these directives at the top of your pages or controls because this is traditionally where developers expect to see them (although the page still compiles if the directives are located at a different place). Of course, you can also add more than a single attribute to your directive statements, as shown in the following:

```
<%@ [Directive] [Attribute=Value] [Attribute=Value] %>
```

The following table describes the directives at your disposal in ASP.NET 3.5.

Directive	Description
`Assembly`	Links an assembly to the Page or user control for which it is associated.
`Control`	Page directive meant for use with user controls (`.ascx`).
`Implements`	Implements a specified .NET Framework interface.
`Import`	Imports specified namespaces into the Page or user control.
`Master`	Enables you to specify master page–specific attributes and values to use when the page parses or compiles. This directive can be used only with master pages (`.master`).
`MasterType`	Associates a class name to a Page in order to get at strongly typed references or members contained within the specified master page.
`OutputCache`	Controls the output caching policies of a Page or user control.
`Page`	Enables you to specify page specific attributes and values to use when the page parses or compiles. This directive can be used only with ASP.NET pages (`.aspx`).
`PreviousPageType`	Enables an ASP.NET page to work with a postback from another page in the application.

Directive	Description
Reference	Links a Page or user control to the current Page or user control.
Register	Associates aliases with namespaces and class names for notation in custom server control syntax.

The following sections provide a quick review of each of these directives.

@Page

The `@Page` directive enables you to specify attributes and values for an ASP.NET page (`.aspx`) to be used when the page is parsed or compiled. This is the most frequently used directive of the bunch. Because the ASP.NET page is such an important part of ASP.NET, you have quite a few attributes at your disposal. The following table summarizes the attributes available through the `@Page` directive.

Attribute	Description
AspCompat	Permits the page to be executed on a single-threaded apartment thread when given a value of `True`. The default setting for this attribute is `False`.
Async	Specifies whether the ASP.NET page is processed synchronously or asynchronously.
AsyncTimeout	Specifies the amount of time in seconds to wait for the asynchronous task to complete. The default setting is 45 seconds. This is a new attribute of ASP.NET 3.5.
AutoEventWireup	Specifies whether the page events are autowired when set to `True`. The default setting for this attribute is `True`.
Buffer	Enables HTTP response buffering when set to `True`. The default setting for this attribute is `True`.
ClassName	Specifies the name of the class that is bound to the page when the page is compiled.
ClientTarget	Specifies the target user agent a control should render content for. This attribute needs to be tied to an alias defined in the `<clientTarget>` section of the web.config.
CodeFile	References the code-behind file with which the page is associated.
CodeFileBaseClass	Specifies the type name of the base class to use with the code-behind class, which is used by the CodeFile attribute.
CodePage	Indicates the code page value for the response.
CompilationMode	Specifies whether ASP.NET should compile the page or not. The available options include `Always` (the default), `Auto`, or `Never`. A setting of Auto means that if possible, ASP.NET will not compile the page.

Attribute	Description
CompilerOptions	Compiler string that indicates compilation options for the page.
CompileWith	Takes a String value that points to the code-behind file used.
ContentType	Defines the HTTP content type of the response as a standard MIME type.
Culture	Specifies the culture setting of the page. ASP.NET 3.5 includes the capability to give the Culture attribute a value of Auto to enable automatic detection of the culture required.
Debug	Compiles the page with debug symbols in place when set to True.
Description	Provides a text description of the page. The ASP.NET parser ignores this attribute and its assigned value.
EnableEventValidation	Specifies whether to enable validation of events in postback and callback scenarios. The default setting of True means that events will be validated.
EnableSessionState	Session state for the page is enabled when set to True. The default setting is True.
EnableTheming	Page is enabled to use theming when set to True. The default setting for this attribute is True.
EnableViewState	View state is maintained across the page when set to True. The default value is True.
EnableViewStateMac	Page runs a machine-authentication check on the page's view state when the page is posted back from the user when set to True. The default value is False.
ErrorPage	Specifies a URL to post to for all unhandled page exceptions.
Explicit	Visual Basic Explicit option is enabled when set to True. The default setting is False.
Language	Defines the language being used for any inline rendering and script blocks.
LCID	Defines the locale identifier for the Web Form's page.
LinePragmas	Boolean value that specifies whether line pragmas are used with the resulting assembly.
MasterPageFile	Takes a String value that points to the location of the master page used with the page. This attribute is used with content pages.
MaintainScrollPositionOn Postback	Takes a Boolean value, which indicates whether the page should be positioned exactly in the same scroll position or if the page should be regenerated in the uppermost position for when the page is posted back to itself.

Attribute	Description
ResponseEncoding	Specifies the response encoding of the page content.
SmartNavigation	Specifies whether to activate the ASP.NET Smart Navigation feature for richer browsers. This returns the postback to the current position on the page. The default value is False.
Src	Points to the source file of the class used for the code behind of the page being rendered.
Strict	Compiles the page using the Visual Basic Strict mode when set to True. The default setting is False.
StylesheetTheme	Applies the specified theme to the page using the ASP.NET 3.5 themes feature. The difference between the StylesheetTheme and Theme attributes is that StylesheetTheme will not override preexisting style settings in the controls, whereas Theme will remove these settings.
Theme	Applies the specified theme to the page using the ASP.NET 3.5 themes feature.
Title	Applies a page's title. This is an attribute mainly meant for content pages that must apply a page title other than what is specified in the master page.
Trace	Page tracing is enabled when set to True. The default setting is False.
TraceMode	Specifies how the trace messages are displayed when tracing is enabled. The settings for this attribute include SortByTime or SortByCategory. The default setting is SortByTime.
Transaction	Specifies whether transactions are supported on the page. The settings for this attribute are Disabled, NotSupported, Supported, Required, and RequiresNew. The default setting is Disabled.
UICulture	The value of the UICulture attribute specifies what UI Culture to use for the ASP.NET page. ASP.NET 3.5 includes the capability to give the UICulture attribute a value of Auto to enable automatic detection of the UICulture.
ValidateRequest	When this attribute is set to True, the form input values are checked against a list of potentially dangerous values. This helps protect your Web application from harmful attacks such as JavaScript attacks. The default value is True.
ViewStateEncryptionMode	Specifies how the ViewState is encrypted on the page. The options include Auto, Always, and Never. The default is Auto.
WarningLevel	Specifies the compiler warning level at which to stop compilation of the page. Possible values are 0 through 4.

Here is an example of how to use the @Page directive:

```
<%@ Page Language="VB" AutoEventWireup="false" CodeFile="Default.aspx.vb"
    Inherits="_Default" %>
```

@Master

The @Master directive is quite similar to the @Page directive except that the @Master directive is meant for master pages (.master). In using the @Master directive, you specify properties of the templated page that you will be using in conjunction with any number of content pages on your site. Any content pages (built using the @Page directive) can then inherit from the master page all the master content (defined in the master page using the @Master directive). Although they are similar, the @Master directive has fewer attributes available to it than does the @Page directive. The available attributes for the @Master directive are shown in the following table.

Attribute	Description
AutoEventWireup	Specifies whether the master page's events are autowired when set to True. Default setting is True.
ClassName	Specifies the name of the class that is bound to the master page when compiled.
CodeFile	References the code-behind file with which the page is associated.
CompilationMode	Specifies whether ASP.NET should compile the page or not. The available options include Always (the default), Auto, or Never. A setting of Auto means that if possible, ASP.NET will not compile the page.
CompilerOptions	Compiler string that indicates compilation options for the master page.
CompileWith	Takes a String value that points to the code-behind file used for the master page.
Debug	Compiles the master page with debug symbols in place when set to True.
Description	Provides a text description of the master page. The ASP.NET parser ignores this attribute and its assigned value.
EnableTheming	Indicates the master page is enabled to use theming when set to True. The default setting for this attribute is True.
EnableViewState	Maintains view state for the master page when set to True. The default value is True.
Explicit	Indicates that the Visual Basic Explicit option is enabled when set to True. The default setting is False.
Inherits	Specifies the CodeBehind class for the master page to inherit.
Language	Defines the language that is being used for any inline rendering and script blocks.

Attribute	Description
LinePragmas	Boolean value that specifies whether line pragmas are used with the resulting assembly.
MasterPageFile	Takes a String value that points to the location of the master page used with the master page. It is possible to have a master page use another master page, which creates a nested master page.
Src	Points to the source file of the class used for the code behind of the master page being rendered.
Strict	Compiles the master page using the Visual Basic Strict mode when set to True. The default setting is False.
WarningLevel	Specifies the compiler warning level at which you want to abort compilation of the page. Possible values are from 0 to 4.

Here is an example of how to use the @Master directive:

```
<%@ Master Language="VB" CodeFile="MasterPage1.master.vb"
    AutoEventWireup="false" Inherits="MasterPage" %>
```

@Control

The @Control directive is similar to the @Page directive except that @Control is used when you build an ASP.NET user control. The @Control directive allows you to define the properties to be inherited by the user control. These values are assigned to the user control as the page is parsed and compiled. The available attributes are fewer than those of the @Page directive, but quite a few of them allow for the modifications you need when building user controls. The following table details the available attributes.

Attribute	Description
AutoEventWireup	Specifies whether the user control's events are autowired when set to True. Default setting is True.
ClassName	Specifies the name of the class that is bound to the user control when the page is compiled.
CodeFileBaseClass	Specifies the type name of the base class to use with the code-behind class, which is used by the CodeFile attribute.
CodeFile	References the code-behind file with which the user control is associated.
CompilerOptions	Compiler string that indicates compilation options for the user control.
CompileWith	Takes a String value that points to the code-behind file used for the user control.
Debug	Compiles the user control with debug symbols in place when set to True.

Attribute	Description
Description	Provides a text description of the user control. The ASP.NET parser ignores this attribute and its assigned value.
EnableTheming	User control is enabled to use theming when set to True. The default setting for this attribute is True.
EnableViewState	View state is maintained for the user control when set to True. The default value is True.
Explicit	Visual Basic Explicit option is enabled when set to True. The default setting is False.
Inherits	Specifies the CodeBehind class for the user control to inherit.
Language	Defines the language used for any inline rendering and script blocks.
LinePragmas	Boolean value that specifies whether line pragmas are used with the resulting assembly.
Src	Points to the source file of the class used for the code behind of the user control being rendered.
Strict	Compiles the user control using the Visual Basic Strict mode when set to True. The default setting is False.
WarningLevel	Specifies the compiler warning level at which to stop compilation of the user control. Possible values are 0 through 4.

The @Control directive is meant to be used with an ASP.NET user control. The following is an example of how to use the directive:

```
<%@ Control Language="VB" Explicit="True"
    CodeFile="WebUserControl.ascx.vb" Inherits="WebUserControl"
    Description="This is the registration user control." %>
```

@Import

The @Import directive allows you to specify a namespace to be imported into the ASP.NET page or user control. By importing, all the classes and interfaces of the namespace are made available to the page or user control. This directive supports only a single attribute: Namespace.

The Namespace attribute takes a String value that specifies the namespace to be imported. The @Import directive cannot contain more than one attribute/value pair. Because of this, you must place multiple namespace imports in multiple lines as shown in the following example:

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
```

Several assemblies are already being referenced by your application. You can find a list of these imported namespaces by looking in the root web.config file found at C:\Windows\Microsoft.NET\Framework\

v2.0.50727\CONFIG. You can find this list of assemblies being referenced from the <assemblies> child element of the <compilation> element. The settings in the root web.config file are as follows:

```
<assemblies>
   <add assembly="mscorlib"/>
   <add assembly="System, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b77a5c561934e089"/>
   <add assembly="System.Configuration, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b03f5f7f11d50a3a"/>
   <add assembly="System.Web, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b03f5f7f11d50a3a"/>
   <add assembly="System.Data, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b77a5c561934e089"/>
   <add assembly="System.Web.Services, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b03f5f7f11d50a3a"/>
   <add assembly="System.Xml, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b77a5c561934e089"/>
   <add assembly="System.Drawing, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b03f5f7f11d50a3a"/>
   <add assembly="System.EnterpriseServices, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b03f5f7f11d50a3a"/>
   <add assembly="System.Web.Mobile, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b03f5f7f11d50a3a"/>
   <add assembly="*"/>
   <add assembly="System.Runtime.Serialization, Version=3.0.0.0, Culture=neutral,
       PublicKeyToken=b77a5c561934e089, processorArchitecture=MSIL"/>
   <add assembly="System.IdentityModel, Version=3.0.0.0, Culture=neutral,
       PublicKeyToken=b77a5c561934e089, processorArchitecture=MSIL"/>
   <add assembly="System.ServiceModel, Version=3.0.0.0, Culture=neutral,
       PublicKeyToken=b77a5c561934e089"/>
   <add assembly="System.ServiceModel.Web, Version=3.5.0.0, Culture=neutral,
       PublicKeyToken=31bf3856ad364e35"/>
   <add assembly="System.WorkflowServices, Version=3.5.0.0, Culture=neutral,
       PublicKeyToken=31bf3856ad364e35"/>
</assemblies>
```

Because of this reference in the root web.config file, these assemblies need not be referenced in a References folder, as you would have done in ASP.NET 1.0/1.1. You can actually add or delete assemblies that are referenced from this list. For example, if you have a custom assembly referenced continuously by each and every application on the server, you can simply add a similar reference to your custom assembly next to these others. Note that you can perform this same task through the application-specific web.config file of your application as well.

Even though assemblies might be referenced, you must still import the namespaces of these assemblies into your pages. The same root web.config file contains a list of namespaces automatically imported into each and every page of your application. This is specified through the <namespaces> child element of the <pages> element.

```
<namespaces>
   <add namespace="System" />
   <add namespace="System.Collections" />
   <add namespace="System.Collections.Specialized" />
   <add namespace="System.Configuration" />
   <add namespace="System.Text" />
```

```
    <add namespace="System.Text.RegularExpressions" />
    <add namespace="System.Web" />
    <add namespace="System.Web.Caching" />
    <add namespace="System.Web.SessionState" />
    <add namespace="System.Web.Security" />
    <add namespace="System.Web.Profile" />
    <add namespace="System.Web.UI" />
    <add namespace="System.Web.UI.WebControls" />
    <add namespace="System.Web.UI.WebControls.WebParts" />
    <add namespace="System.Web.UI.HtmlControls" />
</namespaces>
```

From this XML list, you can see that quite a number of namespaces are imported into each and every one of your ASP.NET pages. Again, you can feel free to modify this selection in the root `web.config` file or even make a similar selection of namespaces from within your application's `web.config` file.

For instance, you can import your own namespace in the `web.config` file of your application in order to make the namespace available on every page where it is utilized.

```
<?xml version="1.0"?>
<configuration>
   <system.web>
      <pages>
         <namespaces>
            <add namespace="MyCompany.Utilities" />
         </namespaces>
      </pages>
   </system.web>
</configuration>
```

Remember that importing a namespace into your ASP.NET page or user control gives you the opportunity to use the classes without fully identifying the class name. For example, by importing the namespace `System.Data.OleDb` into the ASP.NET page, you can refer to classes within this namespace by using the singular class name (`OleDbConnection` instead of `System.Data.OleDb.OleDbConnection`).

@Implements

The `@Implements` directive gets the ASP.NET page to implement a specified .NET Framework interface. This directive supports only a single attribute: `Interface`.

The `Interface` attribute directly specifies the .NET Framework interface. When the ASP.NET page or user control implements an interface, it has direct access to all its events, methods, and properties.

Here is an example of the `@Implements` directive:

```
<%@ Implements Interface="System.Web.UI.IValidator" %>
```

@Register

The `@Register` directive associates aliases with namespaces and class names for notation in custom server control syntax. You can see the use of the `@Register` directive when you drag and drop a user

control onto any of your .aspx pages. Dragging a user control onto the .aspx page causes Visual Studio 2008 to create an @Register directive at the top of the page. This registers your user control on the page so that the control can then be accessed on the .aspx page by a specific name.

The @Register directive supports five attributes, as described in the following table.

Attribute	Description
Assembly	The assembly you are associating with the TagPrefix.
Namespace	The namespace to relate with TagPrefix.
Src	The location of the user control.
TagName	The alias to relate to the class name.
TagPrefix	The alias to relate to the namespace.

Here is an example of how to use the @Register directive to import a user control to an ASP.NET page:

```
<%@ Register TagPrefix="MyTag" Namespace="MyName.MyNamespace"
    Assembly="MyAssembly" %>
```

@Assembly

The @Assembly directive attaches assemblies, the building blocks of .NET applications, to an ASP.NET page or user control as it compiles, thereby making all the assembly's classes and interfaces available to the page. This directive supports two attributes: Name and Src.

- Name: Enables you to specify the name of an assembly used to attach to the page files. The name of the assembly should include the file name only, not the file's extension. For instance, if the file is MyAssembly.vb, the value of the name attribute should be MyAssembly.
- Src: Enables you to specify the source of the assembly file to use in compilation.

The following provides some examples of how to use the @Assembly directive:

```
<%@ Assembly Name="MyAssembly" %>
<%@ Assembly Src="MyAssembly.vb" %>
```

@PreviousPageType

This directive is used to specify the page from which any cross-page postings originate. Cross-page posting between ASP.NET pages is explained later in the section ''Cross-Page Posting'' and again in Chapter 17.

The @PreviousPageType directive is a new directive that works with the new cross-page posting capability that ASP.NET 3.5 provides. This simple directive contains only two possible attributes: TypeName and VirtualPath:

- TypeName: Sets the name of the derived class from which the postback will occur.
- VirtualPath: Sets the location of the posting page from which the postback will occur.

@MasterType

The @MasterType directive associates a class name to an ASP.NET page in order to get at strongly typed references or members contained within the specified master page. This directive supports two attributes:

- TypeName: Sets the name of the derived class from which to get strongly typed references or members.
- VirtualPath: Sets the location of the page from which these strongly typed references and members will be retrieved.

Details of how to use the @MasterType directive are shown in Chapter 8. Here is an example of its use:

```
<%@ MasterType VirtualPath="~/Wrox.master" %>
```

@OutputCache

The @OutputCache directive controls the output caching policies of an ASP.NET page or user control. This directive supports the ten attributes described in the following table.

Attribute	Description
CacheProfile	Allows for a central way to manage an application's cache profile. Use the CacheProfile attribute to specify the name of the cache profile detailed in the web.config.
Duration	The duration of time in seconds that the ASP.NET page or user control is cached.
Location	Location enumeration value. The default is Any. This is valid for .aspx pages only and does not work with user controls (.ascx). Other possible values include Client, Downstream, None, Server, and ServerAndClient.
NoStore	Specifies whether to send a no-store header with the page.
Shared	Specifies whether a user control's output can be shared across multiple pages. This attribute takes a Boolean value and the default setting is false.
SqlDependency	Enables a particular page to use SQL Server cache invalidation.
VaryByControl	Semicolon-separated list of strings used to vary the output cache of a user control.
VaryByCustom	String specifying the custom output caching requirements.
VaryByHeader	Semicolon-separated list of HTTP headers used to vary the output cache.
VaryByParam	Semicolon-separated list of strings used to vary the output cache.

Here is an example of how to use the @OutputCache directive:

```
<%@ OutputCache Duration="180" VaryByParam="None" %>
```

Remember that the Duration attribute specifies the amount of time in *seconds* during which this page is to be stored in the system cache.

@Reference

The `@Reference` directive declares that another ASP.NET page or user control should be compiled along with the active page or control. This directive supports just a single attribute:

- `VirtualPath`: Sets the location of the page or user control from which the active page will be referenced.

Here is an example of how to use the `@Reference` directive:

```
<%@ Reference VirtualPath="~/MyControl.ascx" %>
```

ASP.NET Page Events

ASP.NET developers consistently work with various events in their server-side code. Many of the events that they work with pertain to specific server controls. For instance, if you want to initiate some action when the end user clicks a button on your Web page, you create a button-click event in your server-side code, as shown in Listing 1-6.

Listing 1-6: A sample button-click event shown in VB

```
Protected Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
   Label1.Text = TextBox1.Text
End Sub
```

In addition to the server controls, developers also want to initiate actions at specific moments when the ASP.NET page is being either created or destroyed. The ASP.NET page itself has always had a number of events for these instances. The following list shows you all the page events you could use in ASP.NET 1.0/1.1:

- `AbortTransaction`
- `CommitTransaction`
- `DataBinding`
- `Disposed`
- `Error`
- `Init`
- `Load`
- `PreRender`
- `Unload`

One of the more popular page events from this list is the `Load` event, which is used in VB as shown in Listing 1-7.

Listing 1-7: Using the Page_Load event

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs)
   Handles Me.Load

   Response.Write("This is the Page_Load event")
End Sub
```

Besides the page events just shown, ASP.NET 3.5 has the following events:

- `InitComplete`: Indicates the initialization of the page is completed.
- `LoadComplete`: Indicates the page has been completely loaded into memory.
- `PreInit`: Indicates the moment immediately before a page is initialized.
- `PreLoad`: Indicates the moment before a page has been loaded into memory.
- `PreRenderComplete`: Indicates the moment directly before a page has been rendered in the browser.

An example of using any of these events, such as the `PreInit` event, is shown in Listing 1-8.

Listing 1-8: Using the new page events

VB

```
<script runat="server" language="vb">
    Protected Sub Page_PreInit(ByVal sender As Object, ByVal e As System.EventArgs)
       Page.Theme = Request.QueryString("ThemeChange")
    End Sub
</script>
```

C#

```
<script runat="server">
    protected void Page_PreInit(object sender, System.EventArgs e)
    {
       Page.Theme = Request.QueryString["ThemeChange"];
    }
</script>
```

If you create an ASP.NET 3.5 page and turn on tracing, you can see the order in which the main page events are initiated. They are fired in the following order:

1. `PreInit`
2. `Init`
3. `InitComplete`
4. `PreLoad`
5. `Load`
6. `LoadComplete`

7. `PreRender`
8. `PreRenderComplete`
9. `Unload`

With the addition of these choices, you can now work with the page and the controls on the page at many different points in the page-compilation process. You see these useful new page events in code examples throughout the book.

Dealing with PostBacks

When you are working with ASP.NET pages, be sure you understand the page events just listed. They are important because you place a lot of your page behavior inside these events at specific points in a page lifecycle.

In Active Server Pages 3.0, developers had their pages post to other pages within the application. ASP.NET pages typically post back to themselves in order to process events (such as a button-click event).

For this reason, you must differentiate between posts for the first time a page is loaded by the end user and *postbacks*. A postback is just that — a posting back to the same page. The postback contains all the form information collected on the initial page for processing if required.

Because of all the postbacks that can occur with an ASP.NET page, you want to know whether a request is the first instance for a particular page or is a postback from the same page. You can make this check by using the `IsPostBack` property of the `Page` class, as shown in the following example:

VB

```
If Page.IsPostBack = True Then
   ' Do processing
End If
```

C#

```
if (Page.IsPostBack == true) {
   // Do processing
}
```

In addition to checking against a `True` or `False` value, you can also find out if the request is not a postback in the following manner:

VB

```
If Not Page.IsPostBack Then
   ' Do processing
End If
```

C#

```
if (!Page.IsPostBack) {
   // Do processing
}
```

Cross-Page Posting

One common feature in ASP 3.0 that is difficult to achieve in ASP.NET 1.0/1.1 is the capability to do cross-page posting. Cross-page posting enables you to submit a form (say, `Page1.aspx`) and have this form and all the control values post themselves to another page (`Page2.aspx`).

Traditionally, any page created in ASP.NET 1.0/1.1 simply posted to itself, and you handled the control values within this page instance. You could differentiate between the page's first request and any postbacks by using the `Page.IsPostBack` property, as shown here:

```
If Page.IsPostBack Then
   ' deal with control values
End If
```

Even with this capability, many developers still wanted to be able to post to another page and deal with the first page's control values on that page. This is something that is possible in ASP.NET 3.5, and it is quite a simple process.

For an example, create a page called `Page1.aspx` that contains a simple form. This page is shown in Listing 1-9.

Listing 1-9: Page1.aspx

VB

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    Protected Sub Button1_Click(ByVal sender As Object, _
       ByVal e As System.EventArgs)

        Label1.Text = "Hello " & TextBox1.Text & "<br />" & _
            "Date Selected: " & Calendar1.SelectedDate.ToShortDateString()
    End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>First Page</title>
</head>
<body>
    <form id="form1" runat="server">
        Enter your name:<br />
        <asp:Textbox ID="TextBox1" Runat="server">
        </asp:Textbox>
        <p>
        When do you want to fly?<br />
        <asp:Calendar ID="Calendar1" Runat="server"></asp:Calendar></p>
        <br />
        <asp:Button ID="Button1" Runat="server" Text="Submit page to itself"
```

Continued

```
         OnClick="Button1_Click" />
        <asp:Button ID="Button2" Runat="server" Text="Submit page to Page2.aspx"
         PostBackUrl="~/Page2.aspx" />
        <p>
        <asp:Label ID="Label1" Runat="server"></asp:Label></p>
    </form>
</body>
</html>
```

C#

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

<script runat="server">
    protected void Button1_Click (object sender, System.EventArgs e)
    {
        Label1.Text = "Hello " + TextBox1.Text + "<br />" +
            "Date Selected: " + Calendar1.SelectedDate.ToShortDateString();
    }
</script>
```

The code from `Page1.aspx`, as shown in Listing 1-9, is quite interesting. First, two buttons are shown on the page. Both buttons submit the form, but each submits the form to a different location. The first button submits the form to itself. This is the behavior that has been the default for ASP.NET 1.0/1.1. In fact, nothing is different about `Button1`. It submits to `Page1.aspx` as a postback because of the use of the `OnClick` property in the button control. A `Button1_Click` method on `Page1.aspx` handles the values that are contained within the server controls on the page.

The second button, `Button2`, works quite differently. This button does not contain an `OnClick` method as the first button did. Instead, it uses the `PostBackUrl` property. This property takes a string value that points to the location of the file to which this page should post. In this case, it is `Page2.aspx`. This means that `Page2.aspx` now receives the postback and all the values contained in the `Page1.aspx` controls. Look at the code for `Page2.aspx`, shown in Listing 1-10.

Listing 1-10: Page2.aspx

VB

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs)
        Dim pp_Textbox1 As TextBox
        Dim pp_Calendar1 As Calendar

        pp_Textbox1 = CType(PreviousPage.FindControl("Textbox1"), TextBox)
        pp_Calendar1 = CType(PreviousPage.FindControl("Calendar1"), Calendar)

        Label1.Text = "Hello " & pp_Textbox1.Text & "<br />" & _
            "Date Selected: " & pp_Calendar1.SelectedDate.ToShortDateString()
    End Sub
```

```
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Second Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:Label ID="Label1" Runat="server"></asp:Label>
    </form>
</body>
</html>
```

C#

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    protected void Page_Load(object sender, System.EventArgs e)
    {
        TextBox pp_Textbox1;
        Calendar pp_Calendar1;

        pp_Textbox1 = (TextBox)PreviousPage.FindControl("Textbox1");
        pp_Calendar1 = (Calendar)PreviousPage.FindControl("Calendar1");

        Label1.Text = "Hello " + pp_Textbox1.Text + "<br />" + "Date Selected: " +
           pp_Calendar1.SelectedDate.ToShortDateString();
    }
</script>
```

You have a couple of ways of getting at the values of the controls that are exposed from `Page1.aspx` from the second page. The first option is displayed in Listing 1-10. To get at a particular control's value that is carried over from the previous page, you simply create an instance of that control type and populate this instance using the `FindControl()` method from the `PreviousPage` property. The `String` value assigned to the `FindControl()` method is the `Id` value, which is used for the server control from the previous page. After this is assigned, you can work with the server control and its carried-over values just as if it had originally resided on the current page. You can see from the example that you can extract the `Text` and `SelectedDate` properties from the controls without any problem.

Another way of exposing the control values from the first page (`Page1.aspx`) is to create a `Property` for the control. This is shown in Listing 1-11.

Listing 1-11: Exposing the values of the control from a Property

VB

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

Continued

```
<script runat="server">
   Public ReadOnly Property pp_TextBox1() As TextBox
       Get
           Return TextBox1
       End Get
   End Property

   Public ReadOnly Property pp_Calendar1() As Calendar
       Get
           Return Calendar1
       End Get
   End Property

   Protected Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs)
       Label1.Text = "Hello " & TextBox1.Text & "<br />" & _
           "Date Selected: " & Calendar1.SelectedDate.ToShortDateString()
   End Sub
</script>
```

C#

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    public TextBox pp_TextBox1
    {
        get
        {
            return TextBox1;
        }
    }

    public Calendar pp_Calendar1
    {
        get
        {
            return Calendar1;
        }
    }

    protected void Button1_Click (object sender, System.EventArgs e)
    {
        Label1.Text = "Hello " + TextBox1.Text + "<br />" +
            "Date Selected: " + Calendar1.SelectedDate.ToShortDateString();
    }
</script>
```

Now that these properties are exposed on the posting page, the second page (`Page2.aspx`) can more easily work with the server control properties that are exposed from the first page. Listing 1-12 shows you how `Page2.aspx` works with these exposed properties.

Listing 1-12: Consuming the exposed properties from the first page

VB

```
<%@ Page Language="VB" %>
<%@ PreviousPageType VirtualPath="Page1.aspx" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs)
        Label1.Text = "Hello " & PreviousPage.pp_Textbox1.Text & "<br />" & _
            "Date Selected: " & _
            PreviousPage.pp_Calendar1.SelectedDate.ToShortDateString()
    End Sub
</script>
```

C#

```
<%@ Page Language="C#" %>
<%@ PreviousPageType VirtualPath="Page1.aspx" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    protected void Page_Load(object sender, System.EventArgs e)
    {
        Label1.Text = "Hello " + PreviousPage.pp_TextBox1.Text + "<br />" +
           "Date Selected: " +
           PreviousPage.pp_Calendar1.SelectedDate.ToShortDateString();
    }
</script>
```

In order to be able to work with the properties that `Page1.aspx` exposes, you have to strongly type the `PreviousPage` property to `Page1.aspx`. To do this, you use the `PreviousPageType` directive. This new directive allows you to specifically point to `Page1.aspx` with the use of the `VirtualPath` attribute. When that is in place, notice that you can see the properties that `Page1.aspx` exposes through IntelliSense from the `PreviousPage` property. This is illustrated in Figure 1-7.

As you can see, working with cross-page posting is straightforward. Notice that, when you are cross posting from one page to another, you are not restricted to working only with the postback on the second page. In fact, you can still create methods on `Page1.aspx` that work with the postback before moving onto `Page2.aspx`. To do this, you simply add an `OnClick` event for the button in `Page1.aspx` and a method. You also assign a value for the `PostBackUrl` property. You can then work with the postback on `Page1.aspx` and then again on `Page2.aspx`.

What happens if someone requests `Page2.aspx` before she works her way through `Page1.aspx`? It is actually quite easy to determine if the request is coming from `Page1.aspx` or if someone just hit `Page2.aspx` directly. You can work with the request through the use of the `IsCrossPagePostBack` property that is quite similar to the `IsPostBack` property from ASP.NET 1.0/1.1. The `IsCrossPagePostBack` property enables you to check whether the request is from `Page1.aspx`. Listing 1-13 shows an example of this.

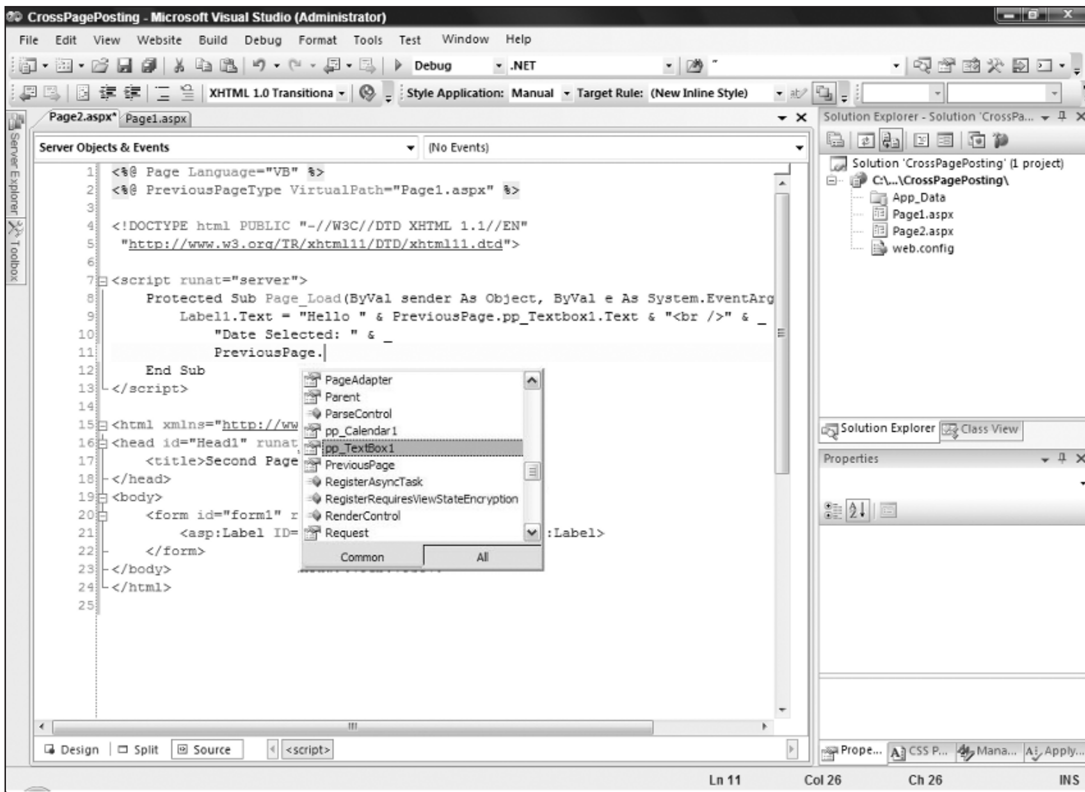

Figure 1-7

Listing 1-13: Using the IsCrossPagePostBack property

VB

```
<%@ Page Language="VB" %>
<%@ PreviousPageType VirtualPath="Page1.aspx" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs)
      If Not PreviousPage Is Nothing And PreviousPage.IsCrossPagePostBack Then
        Label1.Text = "Hello " & PreviousPage.pp_Textbox1.Text & "<br />" & _
            "Date Selected: " & _
            PreviousPage.pp_Calendar1.SelectedDate.ToShortDateString()
      Else
        Response.Redirect("Page1.aspx")
      End If
    End Sub
</script>
```

C#

```
<%@ Page Language="C#" %>
<%@ PreviousPageType VirtualPath="Page1.aspx" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    protected void Page_Load(object sender, System.EventArgs e)
    {
      if (PreviousPage != null && PreviousPage.IsCrossPagePostBack) {
        Label1.Text = "Hello " + PreviousPage.pp_TextBox1.Text + "<br />" +
           "Date Selected: " +
           PreviousPage.pp_Calendar1.SelectedDate.ToShortDateString();
      }
      else
      {
         Response.Redirect("Page1.aspx");
      }
    }
</script>
```

ASP.NET Application Folders

When you create ASP.NET applications, notice that ASP.NET 3.5 uses a file-based approach. When working with ASP.NET, you can add as many files and folders as you want within your application without recompiling each and every time a new file is added to the overall solution. ASP.NET 3.5 includes the capability to automatically precompile your ASP.NET applications dynamically.

ASP.NET 1.0/1.1 compiled everything in your solution into a DLL. This is no longer necessary because ASP.NET applications now have a defined folder structure. By using the ASP.NET defined folders, you can have your code automatically compiled for you, your application themes accessible throughout your application, and your globalization resources available whenever you need them. Look at each of these defined folders to see how they work. The first folder reviewed is the `\App_Code` folder.

\App_Code Folder

The `\App_Code` folder is meant to store your classes, `.wsdl` files, and typed datasets. Any of these items stored in this folder are then automatically available to all the pages within your solution. The nice thing about the `\App_Code` folder is that when you place something inside this folder, Visual Studio 2008 automatically detects this and compiles it if it is a class (`.vb` or `.cs`), automatically creates your XML Web service proxy class (from the `.wsdl` file), or automatically creates a typed dataset for you from your `.xsd` files. After the files are automatically compiled, these items are then instantaneously available to any of your ASP.NET pages that are in the same solution. Look at how to employ a simple class in your solution using the `\App_Code` folder.

The first step is to create an `\App_Code` folder. To do this, simply right-click the solution and choose Add ASP.NET Folder➪App_Code. Right away you will notice that Visual Studio 2008 treats this folder

differently than the other folders in your solution. The `\App_Code` folder is shown in a different color (gray) with a document pictured next to the folder icon. See Figure 1-8.

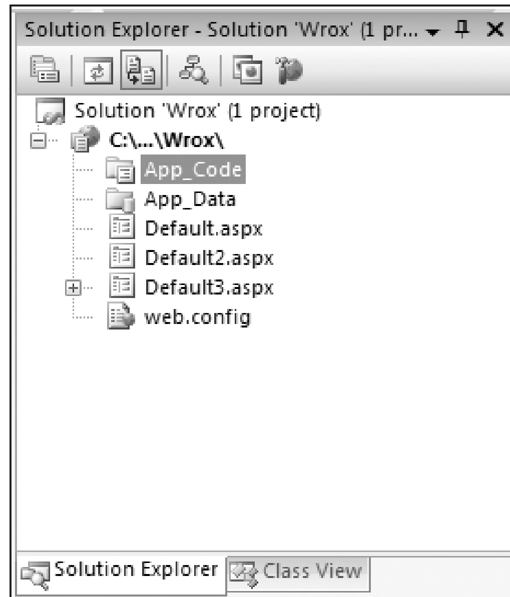

Figure 1-8

After the `\App_Code` folder is in place, right-click the folder and select Add New Item. The Add New Item dialog that appears gives you a few options for the types of files that you can place within this folder. The available options include an AJAX-enabled WCF Service, a Class file, a LINQ to SQL Class, a Text file, a DataSet, a Report, and a Class Diagram if you are using Visual Studio 2008. Visual Web Developer 2008 Express Edition offers only the Class file, Text file, and DataSet file. For the first example, select the file of type Class and name the class `Calculator.vb` or `Calculator.cs`. Listing 1-14 shows how the `Calculator` class should appear.

Listing 1-14: The Calculator class

VB

```
Imports Microsoft.VisualBasic

Public Class Calculator
    Public Function Add(ByVal a As Integer, ByVal b As Integer) As Integer
        Return (a + b)
    End Function
End Class
```

C#

```
using System;

public class Calculator
```

```
{
    public int Add(int a, int b)
        {
            return (a + b);
    }
}
```

What's next? Just save this file, and it is now available to use in any pages that are in your solution. To see this in action, create a simple `.aspx` page that has just a single Label server control. Listing 1-15 shows you the code to place within the `Page_Load` event to make this new class available to the page.

Listing 1-15: An .aspx page that uses the Calculator class

VB

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs)
        Dim myCalc As New Calculator
        Label1.Text = myCalc.Add(12, 12)
    End Sub
</script>
```

C#

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    protected void Page_Load(object sender, System.EventArgs e)
    {
        Calculator myCalc = new Calculator();
        Label1.Text = myCalc.Add(12, 12).ToString();
    }
</script>
```

When you run this `.aspx` page, notice that it utilizes the `Calculator` class without any problem, with no need to compile the class before use. In fact, right after saving the `Calculator` class in your solution or moving the class to the `\App_Code` folder, you also instantaneously receive IntelliSense capability on the methods that the class exposes (as illustrated in Figure 1-9).

To see how Visual Studio 2008 works with the `\App_Code` folder, open the `Calculator` class again in the IDE and add a `Subtract` method. Your class should now appear as shown in Listing 1-16.

Listing 1-16: Adding a Subtract method to the Calculator class

VB

```
Imports Microsoft.VisualBasic
```

Continued

```
Public Class Calculator
    Public Function Add(ByVal a As Integer, ByVal b As Integer) As Integer
        Return (a + b)
    End Function

    Public Function Subtract(ByVal a As Integer, ByVal b As Integer) As Integer
        Return (a - b)
    End Function
End Class
```

C#

```
using System;

public class Calculator
{
    public int Add(int a, int b)
    {
            return (a + b);
    }

    public int Subtract(int a, int b)
    {
            return (a - b);
    }
}
```

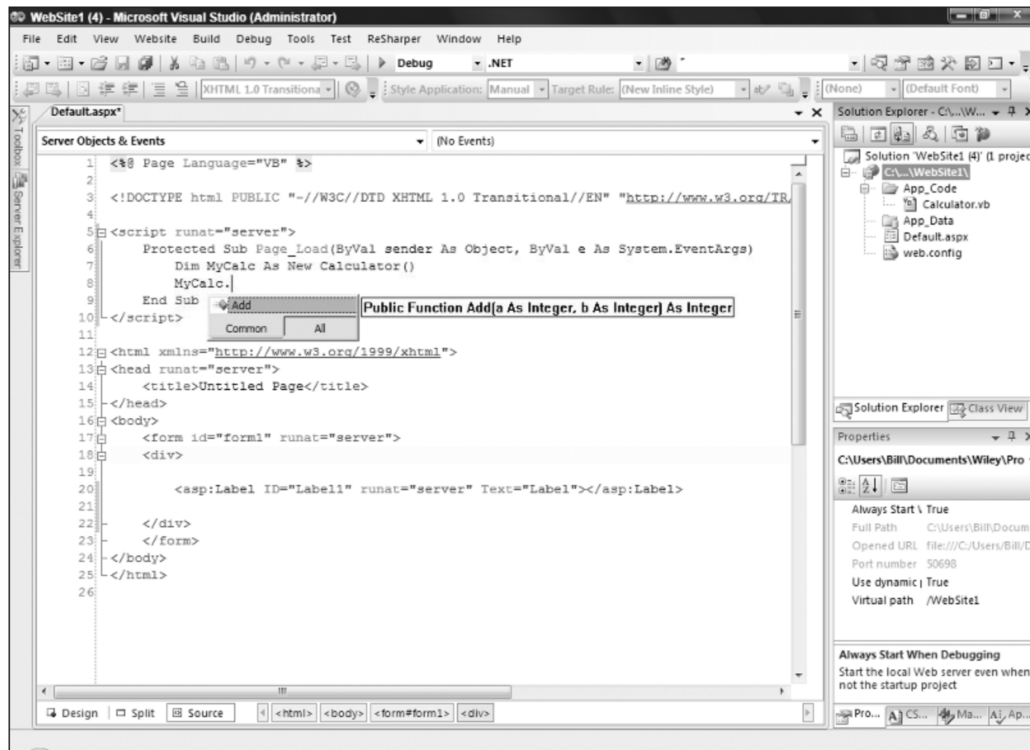

Figure 1-9

After you have added the `Subtract` method to the `Calculator` class, save the file and go back to your `.aspx` page. Notice that the class has been recompiled by the IDE, and the new method is now available to your page. You see this directly in IntelliSense. Figure 1-10 shows this in action.

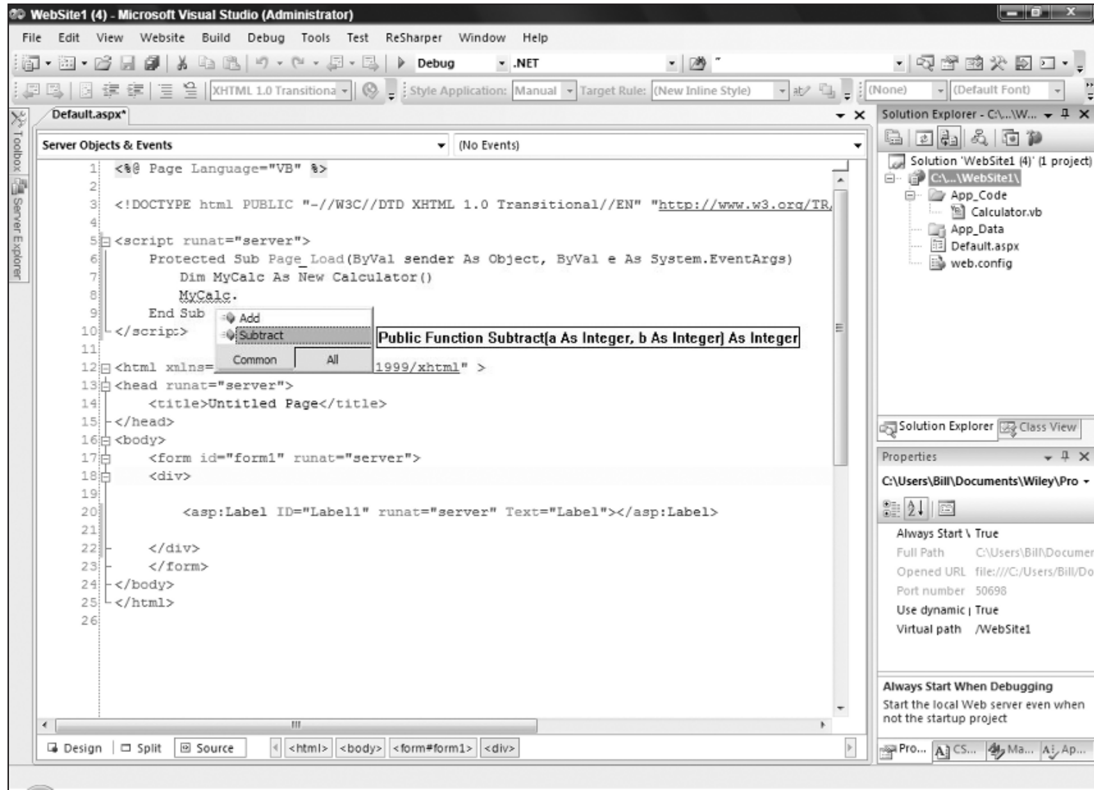

Figure 1-10

Everything placed in the `\App_Code` folder is compiled into a single assembly. The class files placed within the `\App_Code` folder are not required to use a specific language. This means that even if all the pages of the solution are written in Visual Basic 2008, the `Calculator` class in the `\App_Code` folder of the solution can be built in C# (`Calculator.cs`).

Because all the classes contained in this folder are built into a single assembly, you *cannot* have classes of different languages sitting in the root `\App_Code` folder, as in the following example:

```
\App_Code
   Calculator.cs
   AdvancedMath.vb
```

Having two classes made up of different languages in the `\App_Code` folder (as shown here) causes an error to be thrown. It is impossible for the assigned compiler to work with two different languages. Therefore, in order to be able to work with multiple languages in your `\App_Code` folder, you must make some changes to the folder structure and to the `web.config` file.

The first step is to add two new subfolders to the \App_Code folder — a \VB folder and a \CS folder. This gives you the following folder structure:

```
\App_Code
   \VB
      Add.vb
   \CS
      Subtract.cs
```

This still will not correctly compile these class files into separate assemblies, at least not until you make some additions to the web.config file. Most likely, you do not have a web.config file in your solution at this moment, so add one through the Solution Explorer. After it is added, change the <compilation> node so that it is structured as shown in Listing 1-17.

Listing 1-17: Structuring the web.config file so that classes in the \App_Code folder can use different languages

```
<compilation>
   <codeSubDirectories>
      <add directoryName="VB"></add>
      <add directoryName="CS"></add>
   </codeSubDirectories>
</compilation>
```

Now that this is in place in your web.config file, you can work with each of the classes in your ASP.NET pages. In addition, any C# class placed in the CS folder is now automatically compiled just like any of the classes placed in the VB folder. Because you can add these directories in the web.config file, you are not required to name them VB and CS as we did; you can use whatever name tickles your fancy.

\App_Data Folder

The \App_Data folder holds the data stores utilized by the application. It is a good spot to centrally store all the data stores your application might use. The \App_Data folder can contain Microsoft SQL Express files (.mdf files), Microsoft Access files (.mdb files), XML files, and more.

The user account utilized by your application will have read and write access to any of the files contained within the \App_Data folder. By default, this is the ASPNET account. Another reason for storing all your data files in this folder is that much of the ASP.NET system — from the membership and role management systems to the GUI tools, such as the ASP.NET MMC snap-in and ASP.NET Web Site Administration Tool — is built to work with the \App_Data folder.

\App_Themes Folder

Themes are a new way of providing a common look-and-feel to your site across every page. You implement a theme by using a .skin file, CSS files, and images used by the server controls of your site. All these elements can make a *theme*, which is then stored in the \App_Themes folder of your solution. By storing these elements within the \App_Themes folder, you ensure that all the pages within the solution can take advantage of the theme and easily apply its elements to the controls and markup of the page. Themes are discussed in great detail in Chapter 6 of this book.

\App_GlobalResources Folder

Resource files are string tables that can serve as data dictionaries for your applications when these applications require changes to content based on things such as changes in culture. You can add Assembly Resource Files (`.resx`) to the `\App_GlobalResources` folder, and they are dynamically compiled and made part of the solution for use by all your `.aspx` pages in the application. When using ASP.NET 1.0/1.1, you had to use the `resgen.exe` tool and had to compile your resource files to a `.dll` or `.exe` for use within your solution. It is considerably easier to deal with resource files in ASP.NET 3.5. Simply placing your application-wide resources in this folder makes them instantly accessible. Localization is covered in detail in Chapter 31.

\App_LocalResources

Even if you are not interested in constructing application-wide resources using the `\App_GlobalResources` folder, you may want resources that can be used for a single `.aspx` page. You can do this very simply by using the `\App_LocalResources` folder.

You can add resource files that are page-specific to the `\App_LocalResources` folder by constructing the name of the `.resx` file in the following manner:

- `Default.aspx.resx`
- `Default.aspx.fi.resx`
- `Default.aspx.ja.resx`
- `Default.aspx.en-gb.resx`

Now, the resource declarations used on the `Default.aspx` page are retrieved from the appropriate file in the `\App_LocalResources` folder. By default, the `Default.aspx.resx` resource file is used if another match is not found. If the client is using a culture specification of `fi-FI` (Finnish), however, the `Default.aspx.fi.resx` file is used instead. Localization of local resources is covered in detail in Chapter 30.

\App_WebReferences

The `\App_WebReferences`folder is a new name for the previous `Web References` folder used in previous versions of ASP.NET. Now you can use the `\App_WebReferences` folder and have automatic access to the remote Web services referenced from your application. Web services in ASP.NET are covered in Chapter 30.

\App_Browsers

The `\App_Browsers` folder holds `.browser` files, which are XML files used to identity the browsers making requests to the application and understanding the capabilities these browsers have. You can find a list of globally accessible `.browser` files at `C:\Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG\Browsers`. In addition, if you want to change any part of these default browser definition files, just copy the appropriate `.browser` file from the `Browsers` folder to your application's `\App_Browsers` folder and change the definition.

Compilation

You already saw how Visual Studio 2008 compiles pieces of your application as you work with them (for instance, by placing a class in the `\App_Code` folder). The other parts of the application, such as the `.aspx` pages, can be compiled just as they were in earlier versions of ASP.NET by referencing the pages in the browser.

When an ASP.NET page is referenced in the browser for the first time, the request is passed to the ASP.NET parser that creates the class file in the language of the page. It is passed to the ASP.NET parser based on the file's extension (`.aspx`) because ASP.NET realizes that this file extension type is meant for its handling and processing. After the class file has been created, the class file is compiled into a DLL and then written to the disk of the Web server. At this point, the DLL is instantiated and processed, and an output is generated for the initial requester of the ASP.NET page. This is detailed in Figure 1-11.

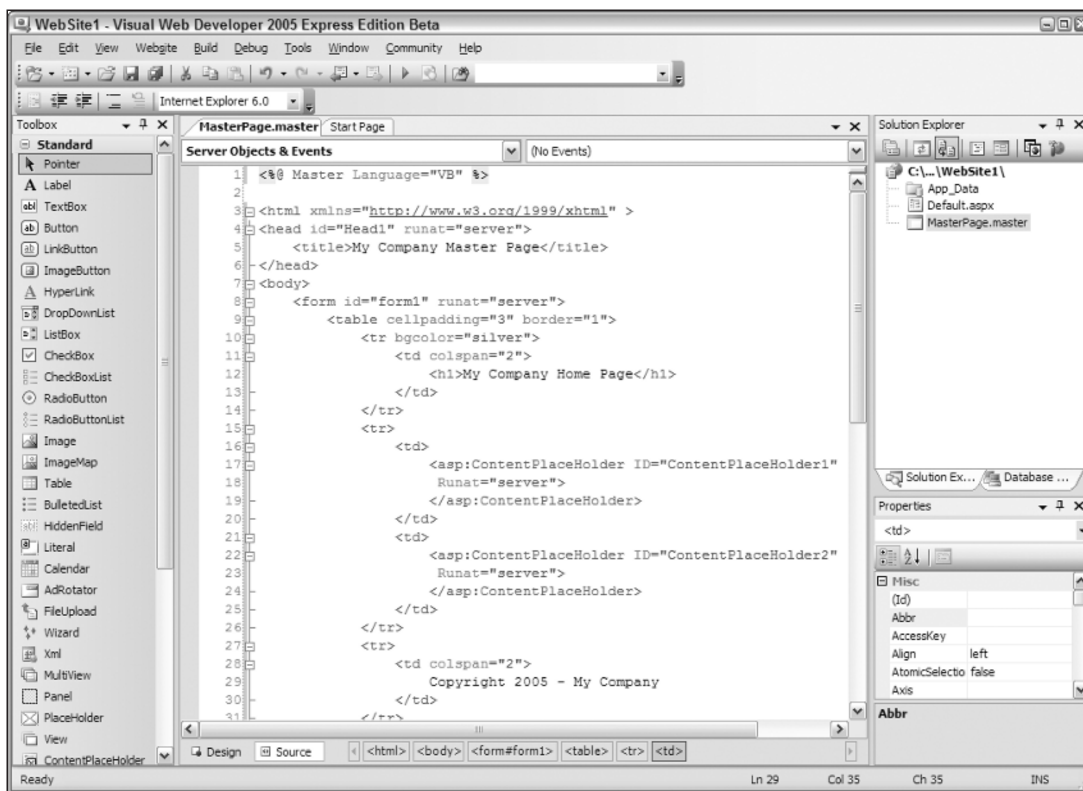

Figure 1-11

On the next request, great things happen. Instead of going through the entire process again for the second and respective requests, the request simply causes an instantiation of the already-created DLL, which sends out a response to the requester. This is illustrated in Figure 1-12.

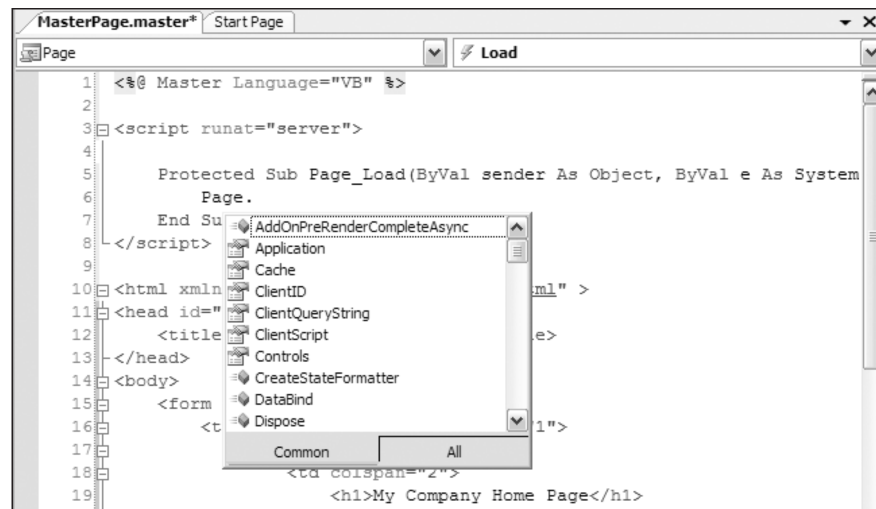

Figure 1-12

Because of the mechanics of this process, if you made changes to your `.aspx` code-behind pages, you found it necessary to recompile your application. This was quite a pain if you had a larger site and did not want your end users to experience the extreme lag that occurs when an `.aspx` page is referenced for the first time after compilation. Many developers, consequently, began to develop their own tools that automatically go out and hit every single page within their application to remove this first-time lag hit from the end user's browsing experience.

ASP.NET 3.5 provides a few ways to precompile your entire application with a single command that you can issue through a command line. One type of compilation is referred to as *in-place precompilation*. In order to precompile your entire ASP.NET application, you must use the `aspnet_compiler.exe` tool that now comes with ASP.NET 3.5. You navigate to the tool using the Command window. Open the Command window and navigate to `C:\Windows\Microsoft.NET\Framework\v2.0.50727\`. When you are there, you can work with the `aspnet_compiler` tool. You can also get to this tool directly by pulling up the Visual Studio 2008 Command Prompt. Choose Start➪All Programs➪Microsoft Visual Studio 2008➪Visual Studio Tools➪Visual Studio 2008 Command Prompt.

After you get the command prompt, you use the aspnet_compiler.exe tool to perform an in-place precompilation using the following command:

```
aspnet_compiler -p "C:\Inetpub\wwwroot\WROX" -v none
```

You then get a message stating that the precompilation is successful. The other great thing about this precompilation capability is that you can also use it to find errors on any of the ASP.NET pages in your application. Because it hits each and every page, if one of the pages contains an error that won't be triggered until runtime, you get notification of the error immediately as you employ this precompilation method.

The next precompilation option is commonly referred to as *precompilation for deployment*. This is an outstanding capability of ASP.NET that enables you to compile your application down to some DLLs, which can then be deployed to customers, partners, or elsewhere for your own use. Not only are minimal steps

required to do this, but also after your application is compiled, you simply have to move around the DLL and some placeholder files for the site to work. This means that your Web site code is completely removed and placed in the DLL when deployed.

However, before you take these precompilation steps, create a folder in your root drive called, for example, `Wrox`. This folder is the one to which you will direct the compiler output. When it is in place, you can return to the compiler tool and give the following command:

```
aspnet_compiler -v [Application Name] -p [Physical Location] [Target]
```

Therefore, if you have an application called `INETA` located at `C:\Websites\INETA`, you use the following commands:

```
aspnet_compiler -v /INETA -p C:\Websites\INETA C:\Wrox
```

Press the Enter key, and the compiler either tells you that it has a problem with one of the command parameters or that it was successful (shown in Figure 1-13). If it was successful, you can see the output placed in the target directory.

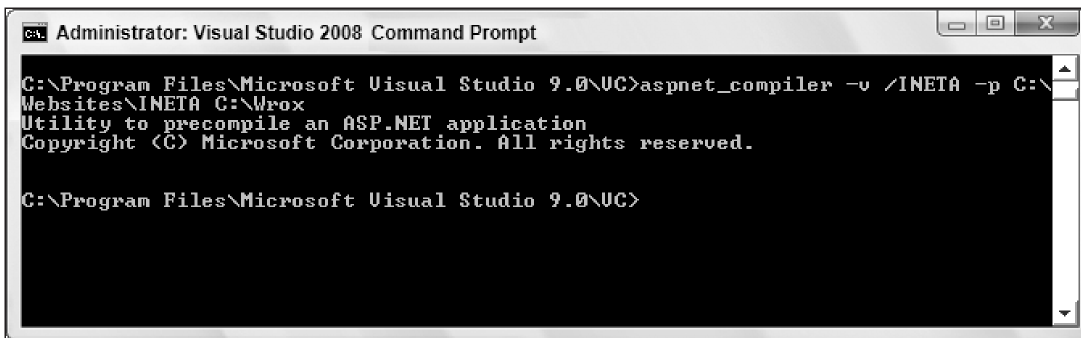

Figure 1-13

In the example just shown, `-v` is a command for the virtual path of the application, which is provided by using `/INETA`. The next command is `-p`, which is pointing to the physical path of the application. In this case, it is `C:\Websites\INETA`. Finally, the last bit, `C:\Wrox`, is the location of the compiler output. The following table describes some of the possible commands for the `aspnet_compiler.exe` tool.

Command	Description
`-m`	Specifies the full IIS metabase path of the application. If you use the `-m` command, you cannot use the `-v` or `-p` command.
`-v`	Specifies the virtual path of the application to be compiled. If you also use the `-p` command, the physical path is used to find the location of the application.

Command	Description
`-p`	Specifies the physical path of the application to be compiled. If this is not specified, the IIS metabase is used to find the application.
`-u`	If this command is utilized, it specifies that the application is updatable.
`-f`	Specifies to overwrite the target directory if it already exists.
`-d`	Specifies that the debug information should be excluded from the compilation process.
`[targetDir]`	Specifies the target directory where the compiled files should be placed. If this is not specified, the output files are placed in the application directory.

After compiling the application, you can go to `C:\Wrox` to see the output. Here, you see all the files and the file structures that were in the original application. However, if you look at the content of one of the files, notice that the file is simply a placeholder. In the actual file, you find the following comment:

```
This is a marker file generated by the precompilation tool
and should not be deleted!
```

In fact, you find a `Code.dll` file in the `bin` folder where all the page code is located. Because it is in a DLL file, it provides great code obfuscation as well. From here on, all you do is move these files to another server using FTP or Windows Explorer, and you can run the entire Web application from these files. When you have an update to the application, you simply provide a new set of compiled files. A sample output is displayed in Figure 1-14.

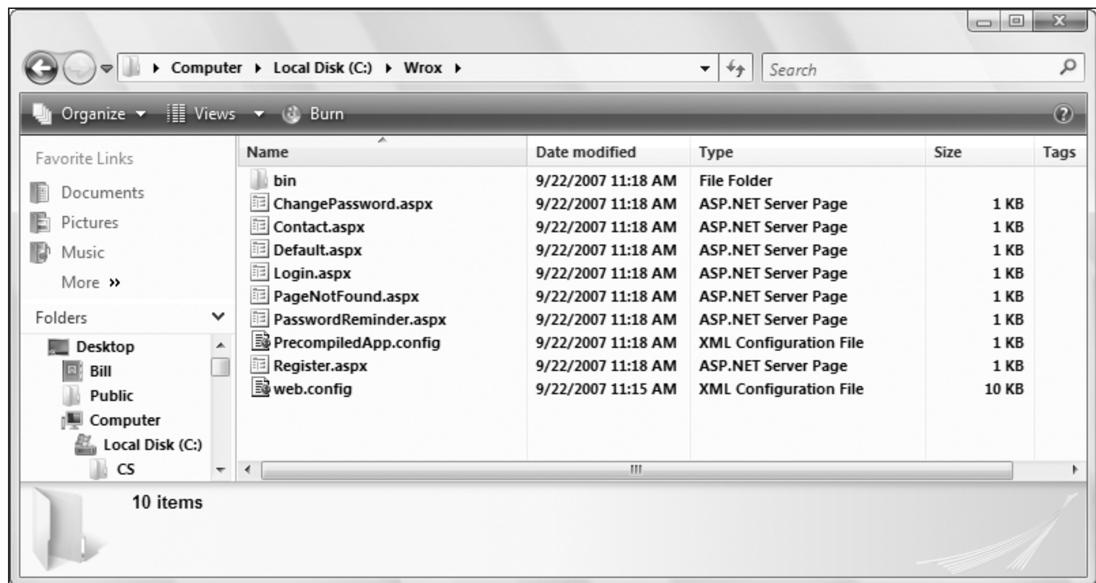

Figure 1-14

Note that this compilation process does not compile *every* type of Web file. In fact, it compiles only the ASP.NET-specific file types and leaves out of the compilation process the following types of files:

- HTML files
- XML files
- XSD files
- `web.config` files
- Text files

You cannot do much to get around this, except in the case of the HTML files and the text files. For these file types, just change the file extensions of these file types to `.aspx`; they are then compiled into the `Code.dll` like all the other ASP.NET files.

Build Providers

As you review the various ASP.NET folders, note that one of the more interesting folders is the \App_Code folder. You can simply drop code files, XSD files, and even WSDL files directly into the folder for automatic compilation. When you drop a class file into the \App_Code folder, the class can automatically be utilized by a running application. In the early days of ASP.NET, if you wanted to deploy a custom component, you had to precompile the component before being able to utilize it within your application. Now ASP.NET simply takes care of all the work that you once had to do. You do not need to perform any compilation routine.

Which file types are compiled in the App_Code folder? As with most things in ASP.NET, this is determined through settings applied in a configuration file. Listing 1-18 shows a snippet of configuration code taken from the master `Web.config` file found in ASP.NET 3.5.

Listing 1-18: Reviewing the list of build providers

```
<compilation>
   <buildProviders>
      <add extension=".aspx" type="System.Web.Compilation.PageBuildProvider" />
      <add extension=".ascx"
       type="System.Web.Compilation.UserControlBuildProvider" />
      <add extension=".master"
       type="System.Web.Compilation.MasterPageBuildProvider" />
      <add extension=".asmx"
       type="System.Web.Compilation.WebServiceBuildProvider" />
      <add extension=".ashx"
       type="System.Web.Compilation.WebHandlerBuildProvider" />
      <add extension=".soap"
       type="System.Web.Compilation.WebServiceBuildProvider" />
      <add extension=".resx" type="System.Web.Compilation.ResXBuildProvider" />
      <add extension=".resources"
       type="System.Web.Compilation.ResourcesBuildProvider" />
      <add extension=".wsdl" type="System.Web.Compilation.WsdlBuildProvider" />
      <add extension=".xsd" type="System.Web.Compilation.XsdBuildProvider" />
      <add extension=".js" type="System.Web.Compilation.ForceCopyBuildProvider" />
```

```
        <add extension=".lic"
         type="System.Web.Compilation.IgnoreFileBuildProvider" />
        <add extension=".licx"
         type="System.Web.Compilation.IgnoreFileBuildProvider" />
        <add extension=".exclude"
         type="System.Web.Compilation.IgnoreFileBuildProvider" />
        <add extension=".refresh"
         type="System.Web.Compilation.IgnoreFileBuildProvider" />
    </buildProviders>
</compilation>
```

This section contains a list of build providers that can be used by two entities in your development cycle. The build provider is first used is during development when you are building your solution in Visual Studio 2008. For instance, placing a `.wsdl` file in the App_Code folder during development in Visual Studio causes the IDE to give you automatic access to the dynamically compiled proxy class that comes from this `.wsdl` file. The other entity that uses the build providers is ASP.NET itself. As stated, simply dragging and dropping a `.wsdl` file in the App_Code folder of a deployed application automatically gives the ASP.NET application access to the created proxy class.

A build provider is simply a class that inherits from `System.Web.Compilation.BuildProvider`. The `<buildProviders>` section in the `Web.config` allows you to list the build provider classes that will be utilized. The capability to dynamically compile any WSDL file is defined by the following line in the configuration file.

```
<add extension=".wsdl" type="System.Web.Compilation.WsdlBuildProvider" />
```

This means that any file utilizing the `.wsdl` file extension is compiled using the `WsdlBuildProvider`, a class that inherits from `BuildProvider`. Microsoft provides a set number of build providers out of the box for you to use. As you can see from the set in Listing 1-18, a number of providers are available in addition to the `WsdlBuildProvider`, including providers such as the `XsdBuildProvider`, `PageBuildProvider`, `UserControlBuildProvider`, `MasterPageBuildProvider`, and more. Just by looking at the names of some of these providers you can pretty much understand what they are about. The next section, however, reviews some other providers whose names might not ring a bell right away.

Using the Built-in Build Providers

Two of the providers that this section covers are the `ForceCopyBuildProvider` and the `IgnoreFileBuildProvider`, both of which are included in the default list of providers.

The `ForceCopyBuildProvider` is basically a provider that copies only those files for deployment that use the defined extension. (These files are not included in the compilation process.) An extension that utilizes the `ForceCopyBuildProvider` is shown in the predefined list in Listing 1-18. This is the `.js` file type (a JavaScript file extension). Any `.js` files are simply copied and not included in the compilation process (which makes sense for JavaScript files). You can add other file types that you want to be a part of this copy process with the command shown here:

```
<add extension=".chm" type="System.Web.Compilation.ForceCopyBuildProvider" />
```

In addition to the `ForceCopyBuildProvider`, you should also be aware of the `IgnoreFileBuildProvider` class. This provider causes the defined file type to be ignored in the deployment or compilation process. This means that any file type defined with `IgnoreFileBuildProvider` is simply ignored. Visual Studio

will not copy, compile, or deploy any file of that type. So, if you are including Visio diagrams in your project, you can simply add the following `<add>` element to the `web.config` file to have this file type ignored. An example is presented here:

```
<add extension=".vsd" type="System.Web.Compilation.IgnoreFileBuildProvider" />
```

With this in place, all `.vsd` files are ignored.

Using Your Own Build Providers

In addition to using the predefined build providers out of the box, you can also take this build provider stuff one-step further and construct your own custom build providers to use within your applications.

For example, suppose you wanted to construct a `Car` class dynamically based upon settings applied in a custom `.car` file that you have defined. You might do this because you are using this `.car` definition file in multiple projects or many times within the same project. Using a build provider makes it simpler to define these multiple instances of the `Car` class.

An example of the `.car` file type is presented in Listing 1-19.

Listing 1-19: An example of a .car file

```
<?xml version="1.0" encoding="utf-8" ?>
<car name="EvjenCar">
  <color>Blue</color>
  <door>4</door>
  <speed>150</speed>
</car>
```

In the end, this XML declaration specifies the name of the class to compile as well as some values for various properties and a method. These elements make up the class. Now that you understand the structure of the `.car` file type, the next step is to construct the build provider. To accomplish this task, create a new Class Library project in the language of your choice within Visual Studio. Name the project `CarBuildProvider`. The `CarBuildProvider` contains a single class — `Car.vb` or `Car.cs`. This class inherits from the base class `BuildProvider` and overrides the `GenerateCode()` method of the `BuildProvider` class. This class is presented in Listing 1-20.

Listing 1-20: The CarBuildProvider

VB

```
Imports System.IO
Imports System.Web.Compilation
Imports System.Xml
Imports System.CodeDom

Public Class Car
    Inherits BuildProvider

    Public Overrides Sub GenerateCode(ByVal myAb As AssemblyBuilder)
        Dim carXmlDoc As XmlDocument = New XmlDocument()
```

```
            Using passedFile As Stream = Me.OpenStream()
                carXmlDoc.Load(passedFile)
            End Using

            Dim mainNode As XmlNode = carXmlDoc.SelectSingleNode("/car")
            Dim selectionMainNode As String = mainNode.Attributes("name").Value

            Dim colorNode As XmlNode = carXmlDoc.SelectSingleNode("/car/color")
            Dim selectionColorNode As String = colorNode.InnerText

            Dim doorNode As XmlNode = carXmlDoc.SelectSingleNode("/car/door")
            Dim selectionDoorNode As String = doorNode.InnerText

            Dim speedNode As XmlNode = carXmlDoc.SelectSingleNode("/car/speed")
            Dim selectionSpeedNode As String = speedNode.InnerText

            Dim ccu As CodeCompileUnit = New CodeCompileUnit()
            Dim cn As CodeNamespace = New CodeNamespace()
            Dim cmp1 As CodeMemberProperty = New CodeMemberProperty()
            Dim cmp2 As CodeMemberProperty = New CodeMemberProperty()
            Dim cmm1 As CodeMemberMethod = New CodeMemberMethod()

            cn.Imports.Add(New CodeNamespaceImport("System"))

            cmp1.Name = "Color"
            cmp1.Type = New CodeTypeReference(GetType(System.String))
            cmp1.Attributes = MemberAttributes.Public
            cmp1.GetStatements.Add(New CodeSnippetExpression("return """ & _
               selectionColorNode & """"))

            cmp2.Name = "Doors"
            cmp2.Type = New CodeTypeReference(GetType(System.Int32))
            cmp2.Attributes = MemberAttributes.Public
            cmp2.GetStatements.Add(New CodeSnippetExpression("return " & _
               selectionDoorNode))

            cmm1.Name = "Go"
            cmm1.ReturnType = New CodeTypeReference(GetType(System.Int32))
            cmm1.Attributes = MemberAttributes.Public
            cmm1.Statements.Add(New CodeSnippetExpression("return " & _
               selectionSpeedNode))

            Dim ctd As CodeTypeDeclaration = New CodeTypeDeclaration(selectionMainNode)
            ctd.Members.Add(cmp1)
            ctd.Members.Add(cmp2)
            ctd.Members.Add(cmm1)

            cn.Types.Add(ctd)
            ccu.Namespaces.Add(cn)

            myAb.AddCodeCompileUnit(Me, ccu)
        End Sub

    End Class
```

Continued

C#

```
using System.IO;
using System.Web.Compilation;
using System.Xml;
using System.CodeDom;

namespace CarBuildProvider
{
    class Car : BuildProvider
    {
        public override void GenerateCode(AssemblyBuilder myAb)
        {
            XmlDocument carXmlDoc = new XmlDocument();

            using (Stream passedFile = OpenStream())
            {
                carXmlDoc.Load(passedFile);
            }
            XmlNode mainNode = carXmlDoc.SelectSingleNode("/car");
            string selectionMainNode = mainNode.Attributes["name"].Value;

            XmlNode colorNode = carXmlDoc.SelectSingleNode("/car/color");
            string selectionColorNode = colorNode.InnerText;

            XmlNode doorNode = carXmlDoc.SelectSingleNode("/car/door");
            string selectionDoorNode = doorNode.InnerText;

            XmlNode speedNode = carXmlDoc.SelectSingleNode("/car/speed");
            string selectionSpeedNode = speedNode.InnerText;

            CodeCompileUnit ccu = new CodeCompileUnit();
            CodeNamespace cn = new CodeNamespace();
            CodeMemberProperty cmp1 = new CodeMemberProperty();
            CodeMemberProperty cmp2 = new CodeMemberProperty();
            CodeMemberMethod cmm1 = new CodeMemberMethod();

            cn.Imports.Add(new CodeNamespaceImport("System"));

            cmp1.Name = "Color";
            cmp1.Type = new CodeTypeReference(typeof(string));
            cmp1.Attributes = MemberAttributes.Public;
            cmp1.GetStatements.Add(new CodeSnippetExpression("return \"" +
               selectionColorNode + "\""));

            cmp2.Name = "Doors";
            cmp2.Type = new CodeTypeReference(typeof(int));
            cmp2.Attributes = MemberAttributes.Public;
            cmp2.GetStatements.Add(new CodeSnippetExpression("return " +
               selectionDoorNode));

            cmm1.Name = "Go";
            cmm1.ReturnType = new CodeTypeReference(typeof(int));
            cmm1.Attributes = MemberAttributes.Public;
```

Continued

```
            cmm1.Statements.Add(new CodeSnippetExpression("return " +
               selectionSpeedNode));

            CodeTypeDeclaration ctd = new CodeTypeDeclaration(selectionMainNode);
            ctd.Members.Add(cmp1);
            ctd.Members.Add(cmp2);
            ctd.Members.Add(cmm1);

            cn.Types.Add(ctd);
            ccu.Namespaces.Add(cn);

            myAb.AddCodeCompileUnit(this, ccu);
        }
    }
}
```

As you look over the `GenerateCode()` method, you can see that it takes an instance of `AssemblyBuilder`. This `AssemblyBuilder` object is from the `System.Web.Compilation` namespace and, because of this, your Class Library project needs to have a reference to the `System.Web` assembly. With all the various objects used in this `Car` class, you also have to import in the following namespaces:

```
Imports System.IO
Imports System.Web.Compilation
Imports System.Xml
Imports System.CodeDom
```

When you have done this, one of the tasks remaining in the `GenerateCode()` method is loading the `.car` file. Because the .car file is using XML for its form, you are able to load the document easily using the `XmlDocument` object. From there, by using the CodeDom, you can create a class that contains two properties and a single method dynamically. The class that is generated is an abstract representation of what is defined in the provided `.car` file. On top of that, the name of the class is also dynamically driven from the value provided via the name attribute used in the main `<Car>` node of the `.car` file.

The `AssemblyBuilder` instance that is used as the input object then compiles the generated code along with everything else into an assembly.

What does it mean that your ASP.NET project has a reference to the `CarBuildProvider` assembly in its project? It means that you can create a `.car` file of your own definition and drop this file into the App_Code folder. The second you drop the file into the App_Code folder, you have instant programmatic access to the definition specified in the file.

To see this in action, you first need a reference to the build provider in either the server's `machine.config` or your application's `web.config` file. A reference is shown in Listing 1-21.

Listing 1-21: Making a reference to the build provider in the web.config file

```
<configuration>
  <system.web>
    <compilation debug="false">
      <buildProviders>
        <add extension=".car" type="CarBuildProvider.Car"/>
```

Continued

```
            </buildProviders>
        </compilation>
    </system.web>
</configuration>
```

The <buildProviders> element is a child element of the <compilation> element. The <buildProviders> element takes a couple of child elements to add or remove providers. In this case, because you want to add a reference to the custom CarBuildProvider object, you use the <add> element. The <add> element can take two possible attributes — extension and type. You must use both of these attributes. In extension attribute, you define the file extension that this build provider will be associated with. In this case, you use the .car file extension. This means that any file using this file extension is associated with the class defined in the type attribute. The type attribute then takes a reference to the CarBuildProvider class that you built — CarBuildProvider.Car.

With this reference in place, you can create the .car file that was shown earlier in Listing 1-19. Place the created .car file in the App_Code folder. You instantly have access to a dynamically generated class that comes from the definition provided via the file. For example, because I used EvjenCar as the value of the name attribute in the <Car> element, this will be the name of the class generated, and I will find this exact name in IntelliSense as I type in Visual Studio.

If you create an instance of the EvjenCar class, you also find that you have access to the properties and the method that this class exposes. This is shown in Figure 1-15.

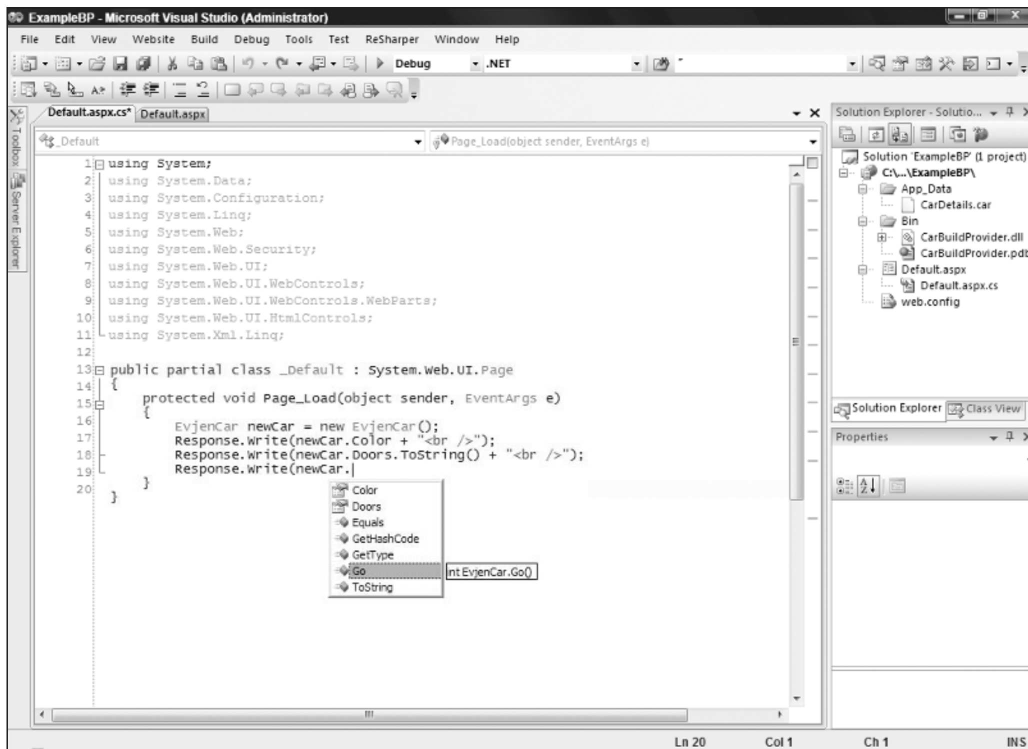

Figure 1-15

In addition to getting access to the properties and methods of the class, you also gain access to the values that are defined in the `.car` file. This is shown in Figure 1-16. The simple code example shown in Figure 1-15 is used for this browser output.

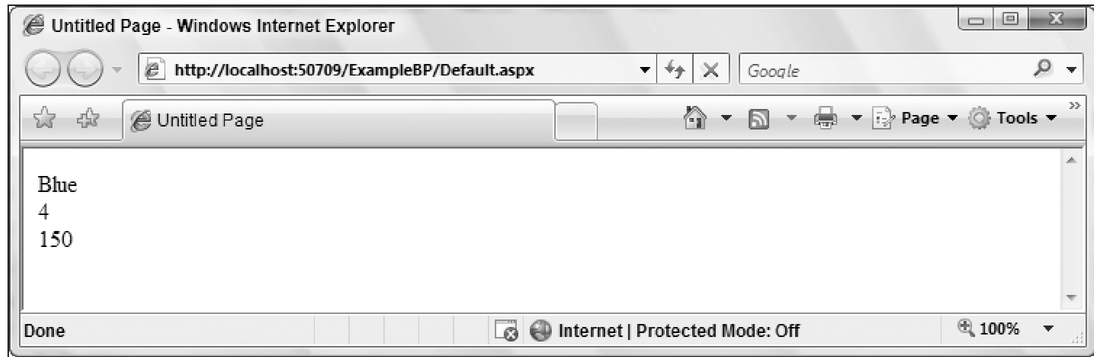

Figure 1-16

Although a car class is not the most useful thing in the world, this example shows you how to take the build provider mechanics into your own hands to extend your application's capabilities.

Global.asax

If you add a new item to your ASP.NET application, you get the Add New Item dialog. From here, you can see that you can add a Global Application Class to your applications. This adds a `Global.asax` file. This file is used by the application to hold application-level events, objects, and variables — all of which are accessible application-wide. Active Server Pages developers had something similar with the `Global.asa` file.

Your ASP.NET applications can have only a single `Global.asax` file. This file supports a number of items. When it is created, you are given the following template:

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

<script runat="server">

    Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
        ' Code that runs on application startup
    End Sub

    Sub Application_End(ByVal sender As Object, ByVal e As EventArgs)
        ' Code that runs on application shutdown
    End Sub

    Sub Application_Error(ByVal sender As Object, ByVal e As EventArgs)
        ' Code that runs when an unhandled error occurs
    End Sub
```

```
    Sub Session_Start(ByVal sender As Object, ByVal e As EventArgs)
        ' Code that runs when a new session is started
    End Sub

    Sub Session_End(ByVal sender As Object, ByVal e As EventArgs)
        ' Code that runs when a session ends.
        ' Note: The Session_End event is raised only when the sessionstate mode
        ' is set to InProc in the Web.config file. If session mode is
        ' set to StateServer
        ' or SQLServer, the event is not raised.
    End Sub

</script>
```

Just as you can work with page-level events in your `.aspx` pages, you can work with overall application events from the `Global.asax` file. In addition to the events listed in this code example, the following list details some of the events you can structure inside this file:

- `Application_Start`: Called when the application receives its very first request. It is an ideal spot in your application to assign any application-level variables or state that must be maintained across all users.
- `Session_Start`: Similar to the `Application_Start` event except that this event is fired when an individual user accesses the application for the first time. For instance, the `Application_Start` event fires once when the first request comes in, which gets the application going, but the `Session_Start` is invoked for each end user who requests something from the application for the first time.
- `Application_BeginRequest`: Although it not listed in the preceding template provided by Visual Studio 2008, the `Application_BeginRequest` event is triggered before each and every request that comes its way. This means that when a request comes into the server, before this request is processed, the `Application_BeginRequest` is triggered and dealt with before any processing of the request occurs.
- `Application_AuthenticateRequest`: Triggered for each request and enables you to set up custom authentications for a request.
- `Application_Error`: Triggered when an error is thrown anywhere in the application by any user of the application. This is an ideal spot to provide application-wide error handling or an event recording the errors to the server's event logs.
- `Session_End`: When running in `InProc` mode, this event is triggered when an end user leaves the application.
- `Application_End`: Triggered when the application comes to an end. This is an event that most ASP.NET developers won't use that often because ASP.NET does such a good job of closing and cleaning up any objects that are left around.

In addition to the global application events that the `Global.asax` file provides access to, you can also use directives in this file as you can with other ASP.NET pages. The `Global.asax` file allows for the following directives:

- `@Application`
- `@Assembly`
- `@Import`

These directives perform in the same way when they are used with other ASP.NET page types.

An example of using the `Global.asax` file is shown in Listing 1-22. It demonstrates how to log when the ASP.NET application domain shuts down. When the ASP.NET application domain shuts down, the ASP.NET application abruptly comes to an end. Therefore, you should place any logging code in the `Application_End` method of the `Global.asax` file.

Listing 1-22: Using the Application_End event in the Global.asax file

VB

```
<%@ Application Language="VB" %>
<%@ Import Namespace="System.Reflection" %>
<%@ Import Namespace="System.Diagnostics" %>

<script runat="server">

    Sub Application_End(ByVal sender As Object, ByVal e As EventArgs)
        Dim MyRuntime As HttpRuntime = _
           GetType(System.Web.HttpRuntime).InvokeMember("_theRuntime", _
           BindingFlags.NonPublic Or BindingFlags.Static Or _
           BindingFlags.GetField, _
           Nothing, Nothing, Nothing)

        If (MyRuntime Is Nothing) Then
            Return
        End If

        Dim shutDownMessage As String = _
           CType(MyRuntime.GetType().InvokeMember("_shutDownMessage", _
           BindingFlags.NonPublic Or BindingFlags.Instance Or
           BindingFlags.GetField, _
           Nothing, MyRuntime, Nothing), System.String)

        Dim shutDownStack As String = _
           CType(MyRuntime.GetType().InvokeMember("_shutDownStack", _
           BindingFlags.NonPublic Or BindingFlags.Instance Or
           BindingFlags.GetField, _
           Nothing, MyRuntime, Nothing), System.String)

        If (Not EventLog.SourceExists(".NET Runtime")) Then
            EventLog.CreateEventSource(".NET Runtime", "Application")
        End If

        Dim logEntry As EventLog = New EventLog()
        logEntry.Source = ".NET Runtime"
        logEntry.WriteEntry(String.Format(_
           "shutDownMessage={0}\r\n\r\n_shutDownStack={1}", _
           shutDownMessage, shutDownStack), EventLogEntryType.Error)
    End Sub

</script>
```

C#

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

Continued

```
<%@ Import Namespace="System.Reflection" %>
<%@ Import Namespace="System.Diagnostics" %>

<script runat="server">

    void Application_End(object sender, EventArgs e)
    {
        HttpRuntime runtime =
           (HttpRuntime)typeof(System.Web.HttpRuntime).InvokeMember("_theRuntime",
           BindingFlags.NonPublic | BindingFlags.Static | BindingFlags.GetField,
           null, null, null);

        if (runtime == null)
        {
            return;
        }

        string shutDownMessage =
           (string)runtime.GetType().InvokeMember("_shutDownMessage",
           BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.GetField,
           null, runtime, null);

        string shutDownStack =
           (string)runtime.GetType().InvokeMember("_shutDownStack",
           BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.GetField,
           null, runtime, null);

        if (!EventLog.SourceExists(".NET Runtime"))
        {
            EventLog.CreateEventSource(".NET Runtime", "Application");
        }

        EventLog logEntry = new EventLog();
        logEntry.Source = ".NET Runtime";
        logEntry.WriteEntry(String.Format("\r\n\r\n_" +
           "shutDownMessage={0}\r\n\r\n_shutDownStack={1}",
           shutDownMessage, shutDownStack), EventLogEntryType.Error);
    }

</script>
```

With this code in place in your `Global.asax` file, start your ASP.NET application. Next, do something to cause the application to restart. You could, for example, make a change to the `web.config` file while the application is running. This triggers the `Application_End` event, and you see the following addition (shown in Figure 1-17) to the event log.

Working with Classes Through VS2008

This chapter showed you how to work with classes within your ASP.NET projects. In constructing and working with classes, you will find that Visual Studio 2008 is quite helpful. Two particularly useful items are a class designer file and an Object Test Bench. The class designer file has an extension of `.cd` and

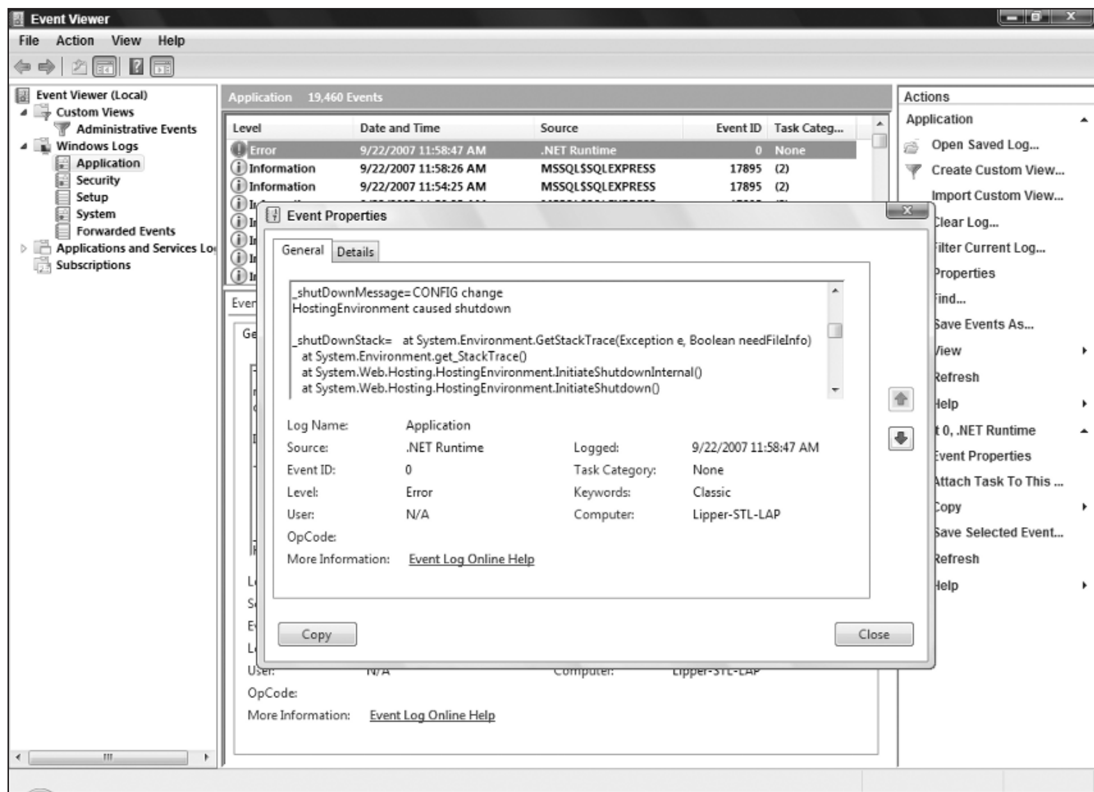

Figure 1-17

gives you a visual way to view your class, as well as all the available methods, properties, and other class items it contains. The Object Test Bench built into Visual Studio gives you a way to instantiate your classes and test them without creating a test application, a task which can be quite time consuming.

To see these items in action, create a new Class Library project in the language of your choice. This project has a single class file, `Class1.vb` or `.cs`. Delete this file and create a new class file called `Calculator.vb` or `.cs`, depending on the language you are using. From here, complete the class by creating a simple `Add()` and `Subtract()` method. Each of these methods takes in two parameters (of type `Integer`) and returns a single Integer with the appropriate calculation performed.

After you have the `Calculator` class in place, the easiest way to create your class designer file for this particular class is to right-click on the `Calculator.vb` file directly in the Solution Explorer and select View Class Diagram from the menu. This creates a `ClassDiagram1.cd` file in your solution.

The visual file, `ClassDiagram1.cd`, is presented in Figure 1-18.

The new class designer file gives you a design view of your class. In the Document Window of Visual Studio, you see a visual representation of the `Calculator` class. The class is represented in a box and

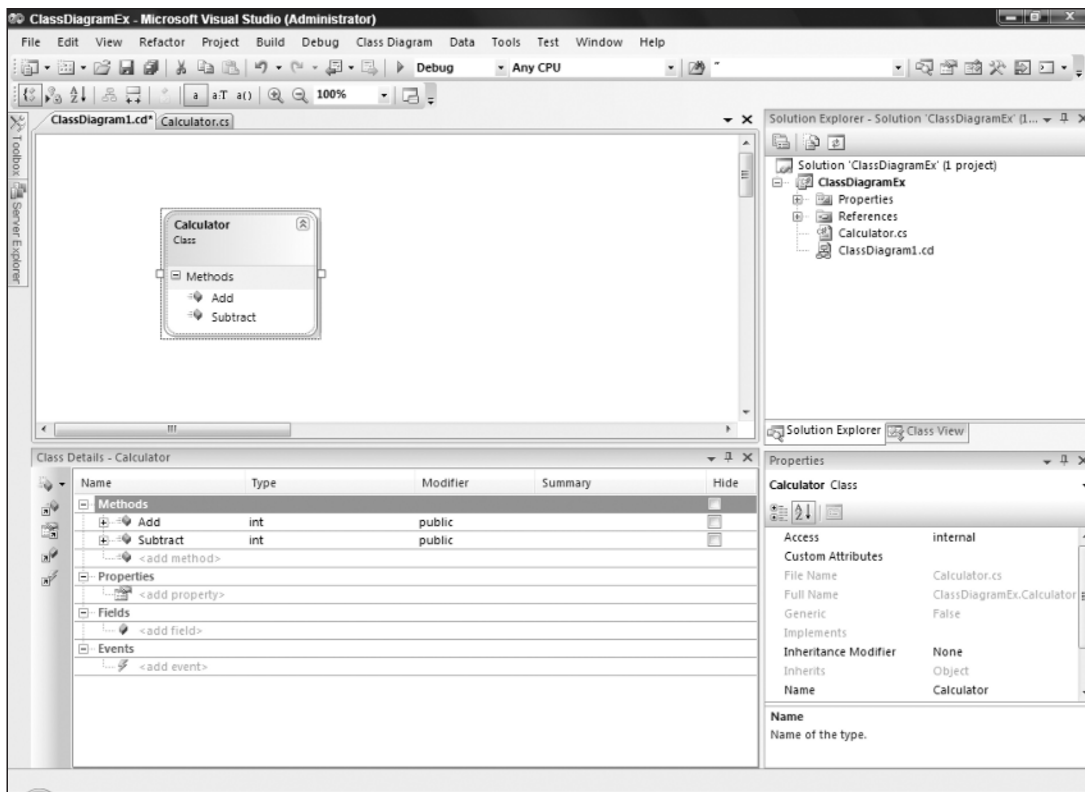

Figure 1-18

provides the name of the class, as well as two available methods that are exposed by the class. Because of the simplicity of this class, the details provided in the visual view are limited.

You can add additional classes to this diagram simply by dragging and dropping class files onto the design surface. You can then arrange the class files on the design surface as you wish. A connection is in place for classes that are inherited from other class files or classes that derive from an interface or abstract class. In fact, you can extract an interface from the class you just created directly in the class designer by right-clicking on the Calculator class box and selecting Refactor ➪ Extract Interface from the provided menu. This launches the Extract Interface dialog that enables you to customize the interface creation. This dialog box is presented in Figure 1-19.

After you click OK, the `ICalculator` interface is created and is then visually represented in the class diagram file, as illustrated in Figure 1-20.

In addition to creating items such as interfaces on-the-fly, you can also modify your `Calculator` class by adding additional methods, properties, events, and more through the Class Details pane found in Visual Studio. The Class Details pane is presented in Figure 1-21.

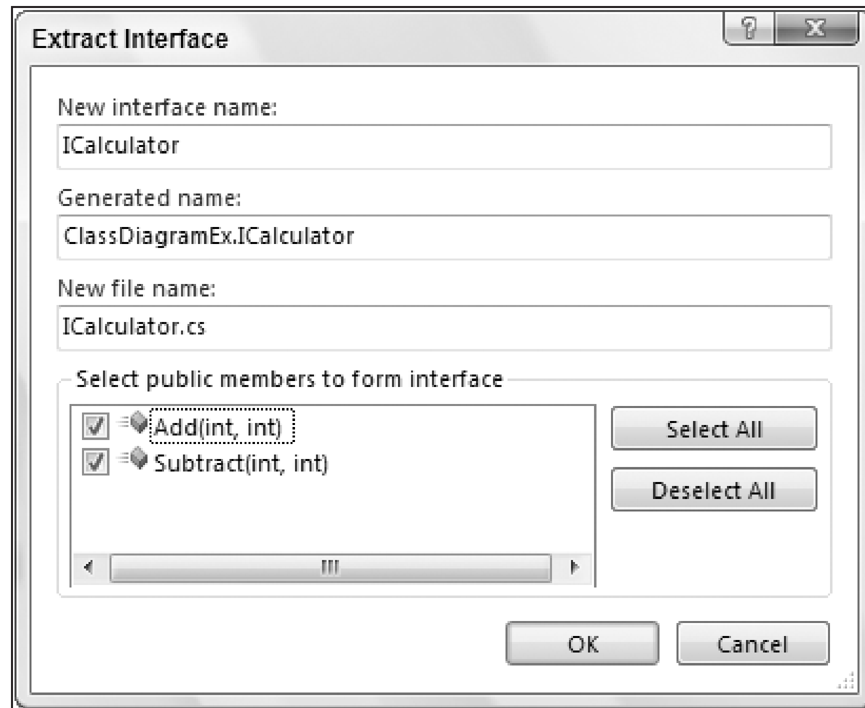

Figure 1-19

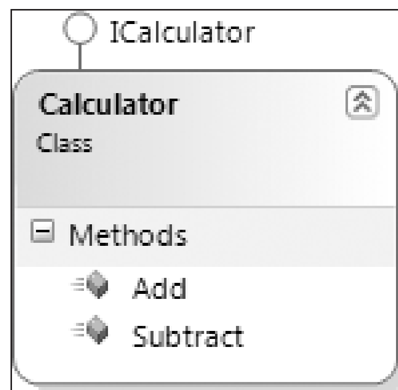

Figure 1-20

From this view of the class, you can directly add any additional methods, properties, fields, or events without directly typing code in your class file. When you enter these items in the Class Details view, Visual Studio generates the code for you on your behalf. For an example of this, add the additional `Multiply()` and `Divide()` methods that the `Calculator` class needs. Expanding the plus sign next to these methods shows the parameters needed in the signature. This is where you add the required `a` and `b` parameters. When you have finished, your Class Details screen should appear as shown in Figure 1-22.

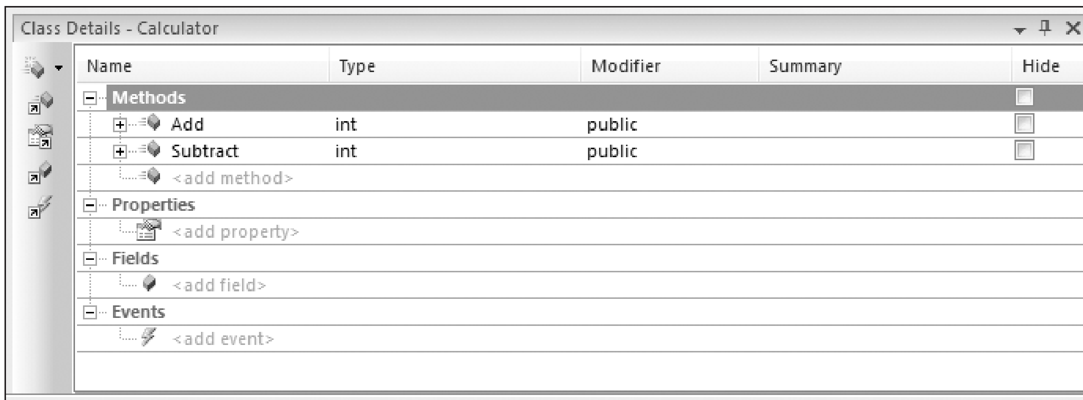

Figure 1-21

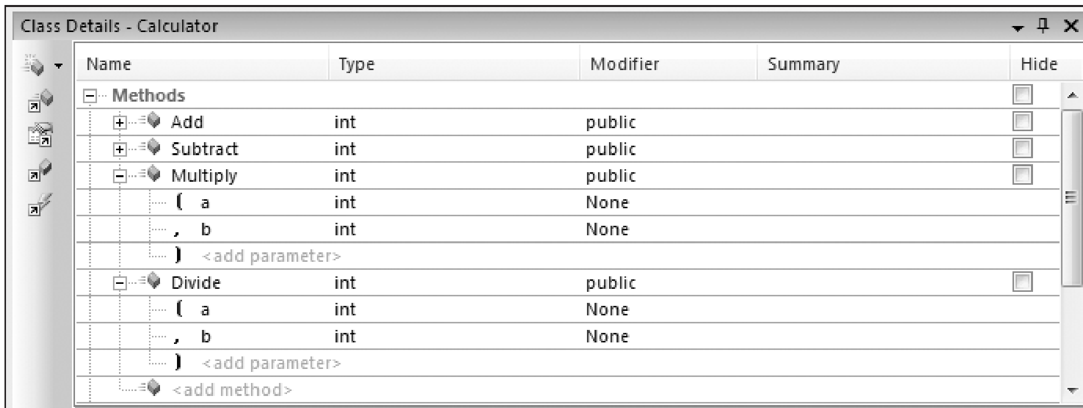

Figure 1-22

After you have added new `Multiply()` and `Divide()` methods and the required parameters, you see that the code in the `Calculator` class has changed to indicate these new methods are present. When the framework of the method is in place, you also see that the class has not been implemented in any fashion. The C# version of the `Multply()` and `Divide()` methods created by Visual Studio is presented in Listing 1-23.

Listing 1-23: The framework provided by Visual Studio's class designer

```
public int Multiply(int a, int b)
{
   throw new System.NotImplementedException();
}

public int Divide(int a, int b)
{
   throw new System.NotImplementedException();
}
```

The new class designer files give you a powerful way to view and understand your classes better — sometimes a picture really is worth a thousand words. One interesting last point on the `.cd` file is that Visual Studio is really doing all the work with this file. If you open the `ClassDesigner1.cd` file in Notepad, you see the results presented in Listing 1-24.

Listing 1-24: The real ClassDesigner1.cd file as seen in Notepad

```
<?xml version="1.0" encoding="utf-8"?>
<ClassDiagram MajorVersion="1" MinorVersion="1">
  <Class Name="ClassDiagramEx.Calculator">
    <Position X="1.25" Y="0.75" Width="1.5" />
    <TypeIdentifier>
      <HashCode>AAIAAAAAAQAAAAAAAAADAAAAAAAAAAAAAAAAAAAAAAA=</HashCode>
      <FileName>Calculator.cs</FileName>
    </TypeIdentifier>
    <Lollipop Position="0.2" />
  </Class>
  <Font Name="Segoe UI" Size="8.25" />
</ClassDiagram>
```

As you can see, it is a rather simple XML file that defines the locations of the class and the items connected to the class.

In addition to using the new class designer to provide a visual representation of your classes, you can also use it to instantiate and test your new objects. To do this, right-click on the `Calculator` class file in the `ClassDiagram1.cd` file and select Create Instance➪Calculator() from the provided menu.

This launches the Create Instance dialog that simply asks you to create a new name for your class instantiation. This dialog is illustrated in Figure 1-23.

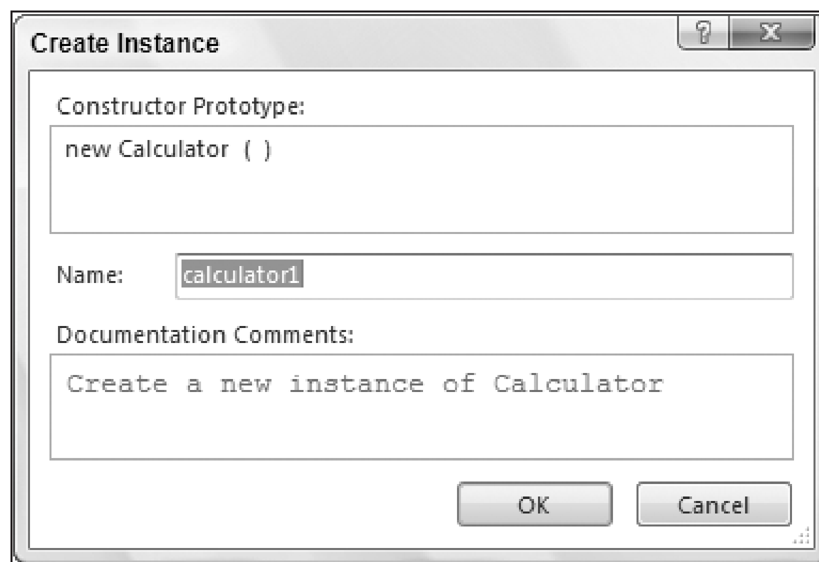

Figure 1-23

From here, click OK and you see a visual representation of this instantiation in the new Object Test Bench directly in Visual Studio. The Object Test Bench now contains only a single gray box classed calculator1. Right-click on this object directly in the Object Test Bench, and select Invoke Method➪Add(int, int) from the provided menu. This is illustrated in Figure 1-24.

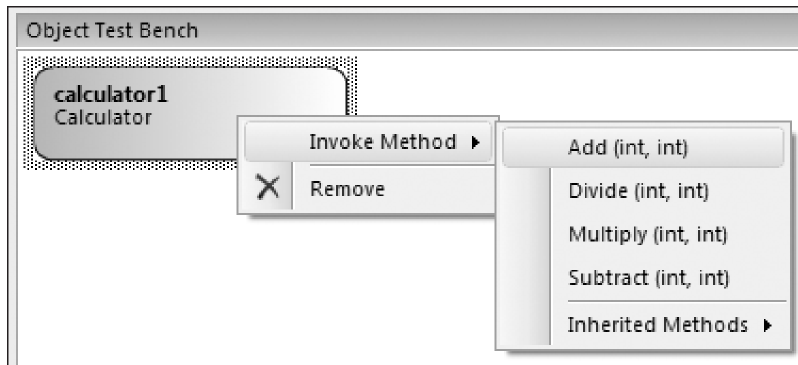

Figure 1-24

Selecting the Add method launches another dialog — the Invoke Method dialog. This dialog enables you to enter values for the required parameters, as shown in Figure 1-25.

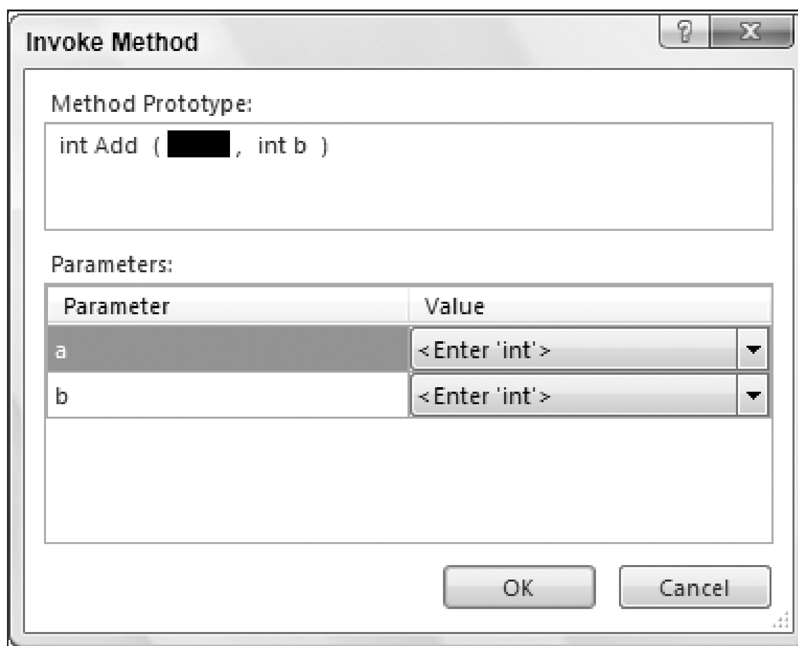

Figure 1-25

After providing values and clicking OK, you see another dialog that provides you with the calculated result, as shown in Figure 1-26.

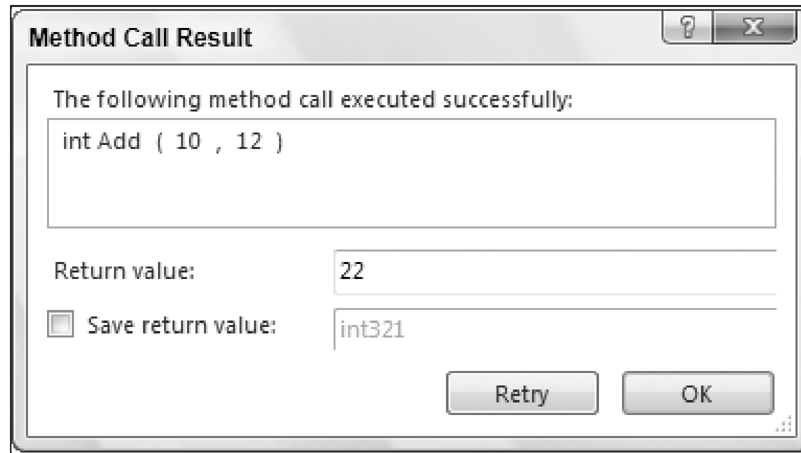

Figure 1-26

This is a simple example. When you start working with more complex objects and collections, however, this feature is even more amazing because the designer enables you to work through the entire returned result visually directly in the IDE.

Summary

This chapter covered a lot of ground. It discussed some of the issues concerning ASP.NET applications as a whole and the choices you have when building and deploying these new applications. With the help of Visual Studio 2008, you now have options about which Web server to use when building your application and whether to work locally or remotely through the new built-in FTP capabilities.

ASP.NET 3.5 and Visual Studio 2008 make it easy to build your pages using an inline coding model or to select a new and better code-behind model that is simpler to use and easier to deploy. You also learned about the new cross-posting capabilities and the new fixed folders that ASP.NET 3.5 has incorporated to make your life easier. These folders make their resources available dynamically with no work on your part. You saw some of the outstanding new compilation options that you have at your disposal. Finally, you looked at ways in which Visual Studio 2008 makes it easy to work with the classes of your project.

As you worked through some of the examples, you may have been thinking, ‘‘WOW!’’ But wait . . . there’s plenty more to come!