



# Incorporating ASP.NET 2.0 into Microsoft Office SharePoint Server 2007

**S**harePoint technologies have been around for a couple of years now and have proven to be quite useful for companies around the world that are implementing portal, team collaboration, or enterprise content management (ECM) strategies.

The development model for working with SharePoint technologies has changed considerably in Microsoft Office SharePoint Server 2007, largely influenced by the release of Microsoft .NET Framework 2.0. This chapter covers the integration between the exciting new features of ASP.NET 2.0 and Microsoft Office SharePoint Server 2007.

First, we will discuss how the integration between Microsoft Office SharePoint Server 2007 and .NET Framework 2.0 works. Then, we will look at the benefits offered by the Visual Studio 2005 Extensions for Windows SharePoint Services 3.0. After that, we will create some web parts using these tools and show you how to use a couple of the new ASP.NET 2.0 server controls within web parts along the way. The chapter finishes with a discussion of the Guidance Automation Toolkit, also known as GAT, and how it can be used to enhance the development experience.

## Architecture Overview

Before looking at Microsoft Office SharePoint Server 2007, we shall take a quick look at its predecessor. SharePoint Products and Technologies 2003 is more or less integrated with IIS 6.0 and ASP.NET 1.1, using an ISAPI filter that intercepts all IIS page requests. In most cases, requests are forwarded to ASP.NET; in other cases, the ISAPI filter determines that a request should be forwarded by an HTTP handler called `SharePointHandler`. In the latter case, SharePoint passes page requests to the `SafeMode` parser, a construct specifically created for SharePoint Products and Technologies 2003. The most important features provided by the `SafeMode` parser are related to advanced caching scenarios and implementing additional security measures. The biggest problem associated with this architecture is that it is not built on top of ASP.NET technology. As a result, the ASP.NET runtime does not always handle all incoming requests first, and therefore does not have the chance to initialize a request with the ASP.NET context. This, along with the use of the custom `SafeMode` parser, has caused problems and confused a lot of the SharePoint 2003 developers out there.

ASP.NET 2.0 was released well after SharePoint Products and Technologies 2003. But even so, you can configure SharePoint Products and Technologies 2003 to support ASP.NET 2.0, as we discussed in detail in the first edition of this book (which can be found on Amazon.com at the following location: [http://www.amazon.com/Pro-SharePoint-2003-Development-Techniques/dp/1590597613/ref=sr\\_1\\_2/102-5827992-1824111?ie=UTF8&s=books&qid=1179750099&sr=8-2](http://www.amazon.com/Pro-SharePoint-2003-Development-Techniques/dp/1590597613/ref=sr_1_2/102-5827992-1824111?ie=UTF8&s=books&qid=1179750099&sr=8-2)).

Microsoft Office SharePoint Server 2007 has been redesigned completely when it comes to integrating with ASP.NET 2.0; it is now built on top of ASP.NET 2.0 and is far more reliant on the features provided by the ASP.NET 2.0 infrastructure.

---

**Note** Configuring SharePoint Products and Technologies 2003 to support ASP.NET 2.0 used to be tricky. In Microsoft Office SharePoint Server 2007, integration is available out of the box. Planning and installing Microsoft Office SharePoint Server 2007 remains a complex topic, but it falls outside the scope of this book. The book *Microsoft SharePoint: Building Office 2007 Solutions in C# 2005* by Scot Hillier (Apress, 2007) contains extensive information about this topic, including capacity planning guidelines, deployment architectures, and detailed instructions on how to create a SharePoint 2007 development environment using Virtual Server 2005.

---

In Microsoft Office SharePoint Server 2007, incoming requests are always handled first by the ASP.NET runtime. The SharePoint ISAPI filter has been removed, and instead the Microsoft Office SharePoint Server 2007 infrastructure relies on ASP.NET 2.0 by defining HTTP modules and HTTP handlers responsible for handling SharePoint-related requests.

---

**Note** All HTTP requests are handled by the ASP.NET engine, which ensures that the ASP.NET context is initialized entirely before SharePoint gets a chance to process a request.

---

In the web.config file of a SharePoint web application, you will notice that the `<httpHandlers>` section contains an HTTP handler called `Microsoft.SharePoint.ApplicationRuntime.SPHttpHandler`. This handler ensures that the `SPHttpHandler` will eventually handle all requests for all file types (for instance, .aspx, .txt, .doc, or .docx files). You can find the web.config file by following these steps:

1. Open a command prompt and type the following command: **inetmgr**. This opens Internet Information Services (IIS) Manager.
2. Expand the [server name] (local computer) node.
3. Expand the Web Sites node.
4. Right-click the SharePoint web application and choose Properties. This opens the [web application] Properties window.
5. Click the Home Directory tab and copy the value of the Local path text field.
6. Open an instance of Windows Explorer and navigate to the path found in the previous step.
7. Open the web.config file found in this folder in any text editor.

When the ASP.NET engine is finished processing a request, it calls the SharePoint Virtual Path Provider (also known as the WSS File provider or `SPVirtualPathProvider`). ASP.NET 2.0 introduces the concept of *virtual path providers*, which are pluggable components that integrate with ASP.NET and can be used to parse .aspx pages. The SharePoint Virtual Path Provider is able to retrieve .aspx pages from a SQL Server database with full-text extensions and does not suffer from the limited functionality regarding page parsing found in SharePoint 2003.

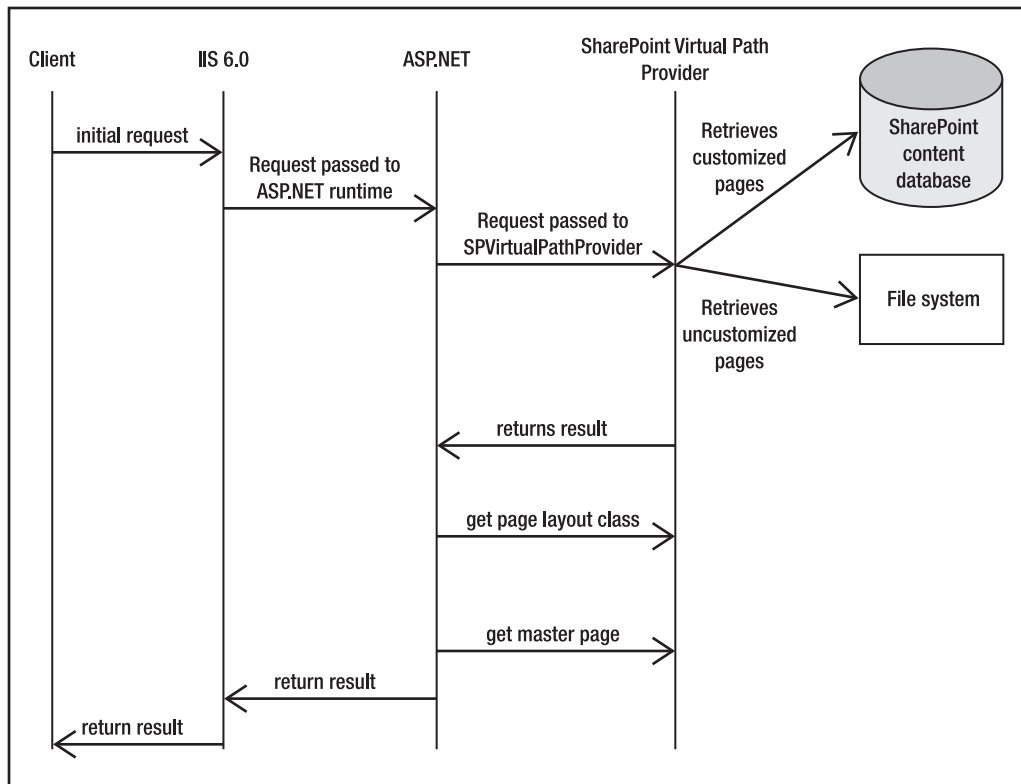
Microsoft Office SharePoint Server 2007 supports two types of pages: uncustomized and customized.

Uncustomized pages are page templates that are located on the file system of the web server. Customized pages are modified page templates that are written to a SharePoint content database. One task the SharePoint Virtual Path Provider is responsible for when handling a page request is

deciding whether the page is located on the file system or in a content database. The SafeMode parser optionally parses pages that are retrieved from the database.

**Note** Microsoft Office 2007 SharePoint Designer can be used to customize pages or undo customizations (also known as reverting back to the original template). In SharePoint Products and Technologies 2003, customized pages were called *unghosted pages*; uncustomized pages were called *ghosted pages*.

After the SharePoint Virtual Path Provider has had the opportunity to handle (but not compile) a page, it returns the result to the ASP.NET engine that compiles the page, which then asks the virtual path provider to fetch the page layout class, which is also compiled. The ASP.NET engine adds SharePoint context data to the web metadata and then retrieves the master page associated to the SharePoint page. The master page is compiled, and a response is returned to the client. Figure 1-1 shows an overview of the page-handling process in Microsoft Office SharePoint Server 2007.



**Figure 1-1.** Handling a page request in Microsoft Office SharePoint Server 2007

## Web Parts Overview

After the huge success of web parts in SharePoint technologies, web parts have been incorporated into ASP.NET 2.0. This has had an effect on Microsoft Office SharePoint Server 2007, which supports

the use of ASP.NET 2.0 web parts. There is also support for SharePoint legacy web parts, accomplished by rebasing the web part base class, which will be discussed later in this section.

Although support for SharePoint 2003 web parts is still available primarily for legacy reasons, they are not the only reason. The web part connection model in ASP.NET 2.0 is in some ways inferior to the connection model for SharePoint legacy web parts. Because of this, you may want to use SharePoint legacy web parts instead of ASP.NET 2.0 web parts. You might want to use SharePoint legacy web parts when you want to

- Create cross-page connections. Differences in the web part connection model will be discussed in Chapter 5.
- Use web part caching.
- Communicate with web parts outside web part zones.

Other than that, it is strongly recommended that you use ASP.NET 2.0 web parts when creating new web parts.

SharePoint legacy web parts all inherit from the `Microsoft.SharePoint.WebPartPages.WebPart` class, the base class for all SharePoint legacy web parts. Listing 1-1 shows a SharePoint legacy web part that overrides the `CreateChildControls()` method and uses generics (one of the new features of the .NET Framework 2.0) to print a “Hello World” message to the page.

**Listing 1-1. Hello World Web Part**

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using Microsoft.SharePoint.WebPartPages;

namespace LoisAndClark.WPLibrary
{
    ..public class MyWP : WebPart
    {
        protected override void CreateChildControls()
        {
            Content obj = new Content();
            string str1 = obj.MyContent<string>("Hello World!");
            this.Controls.Add(new System.Web.UI.LiteralControl(str1));
        }
    }
}
```

For the purposes of this example, we have created a new class called `Content` that is used in the SharePoint legacy web part. Listing 1-2 shows the `Content` class.

**Listing 1-2.** *Example Class Using Generics*

```
namespace LoisAndClark.WPLibrary
{
    public class Content
    {
        internal Content() { }

        public string MyContent<MyType>(MyType arg)
        {
            return arg.ToString();
        }
    }
}
```

All ASP.NET 2.0 web parts inherit from the `System.Web.UI.WebControls.WebParts.WebPart` base class. Listing 1-3 shows an ASP.NET 2.0 web part that writes a “Hello World” message to the page that contains the web part.

**Listing 1-3.** *The Complete Code for a Hello World Web Part*

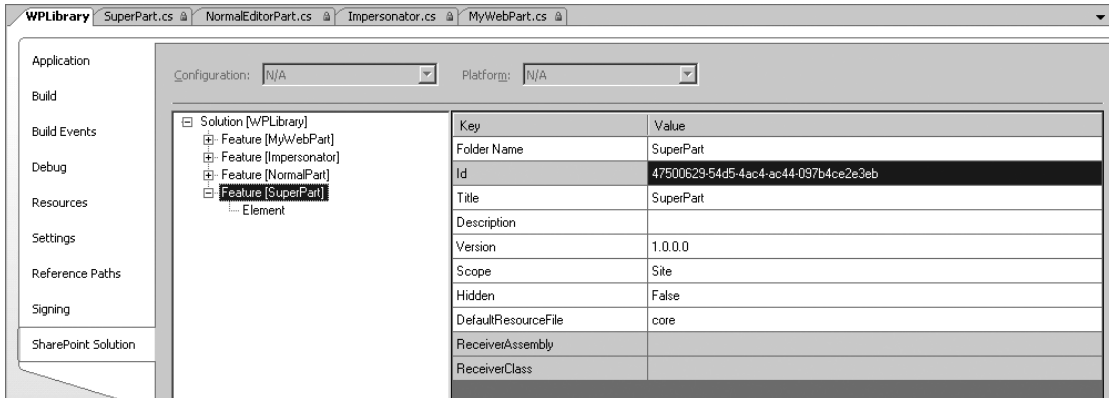
```
using System;
using System.Runtime.InteropServices;
using System.Web.UI;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Serialization;

using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;
using Microsoft.SharePoint.WebPartPages;

namespace LoisAndClark.WPLibrary
{
    [Guid(“c22d5295-81a3-4a23-97ed-3a0a3e8caecf”)]
    public class WPLibrary : System.Web.UI.WebControls.WebParts.WebPart
    {
        public WPLibrary()
        {
            this.ExportMode = WebPartExportMode.All;
        }

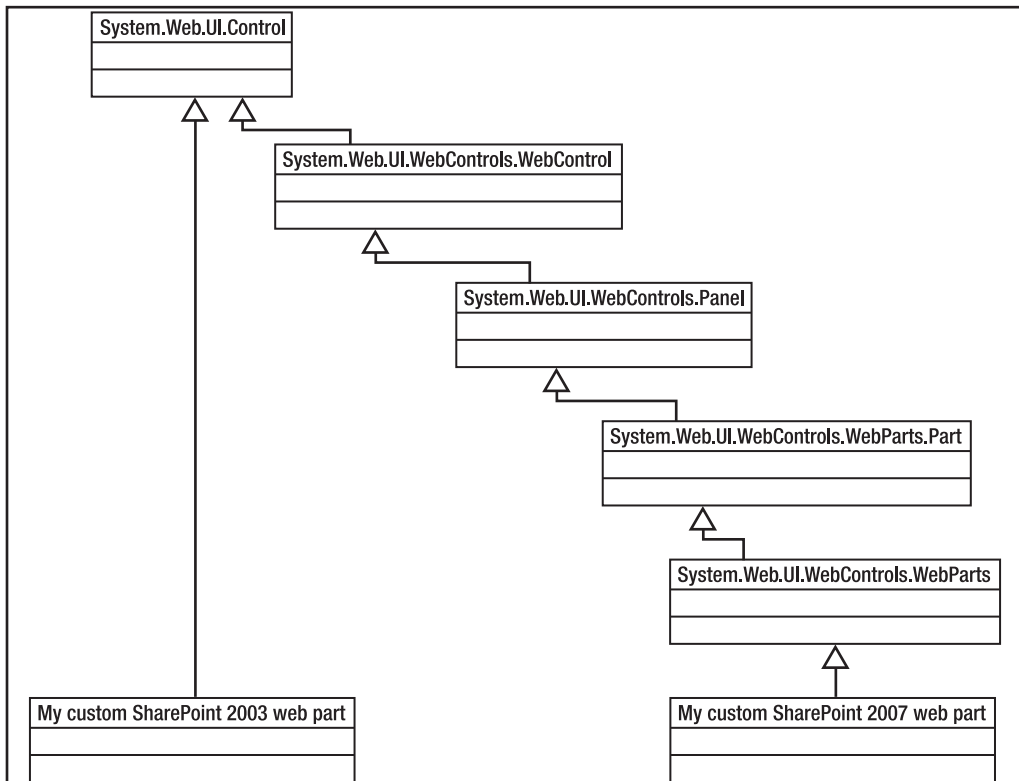
        protected override void Render(HtmlTextWriter writer)
        {
            writer.Write(“Hello World!”);
        }
    }
}
```

Notice the presence of the GUID class attribute. This attribute explicitly (instead of automatically) assigns a GUID to this class. Including this GUID is necessary; otherwise web part deployment via Visual Studio 2005 extensions for Windows SharePoint Services 3.0 fails. If you right-click a web part library that was created using Visual Studio 2005 extensions for Windows SharePoint Services 3.0 in Visual Studio 2005 and click the SharePoint Solution tab, you will see a solution called Solution ([web part library name]). (Solutions are discussed in detail in the section “Deploying a Web Part,” later in this chapter.) This solution contains a separate feature for every web part, as can be seen in Figure 1-2. During deployment, the class GUID of every web part is used as the GUID for a feature containing a web part.



**Figure 1-2.** Viewing SharePoint Solution properties

In order to be able to support both SharePoint legacy web parts and ASP.NET 2.0, the `Microsoft.SharePoint.WebPartPage.WebPart` class is rebased. In SharePoint 2003, this class used to inherit from the `System.Web.UI.Control` class; in Microsoft Office SharePoint Server 2007, the `Microsoft.SharePoint.WebPartPage.WebPart` class inherits from the `System.Web.UI.WebControls.WebParts.WebPart` class. This is shown in Figure 1-3.



**Figure 1-3.** Rebasing the SharePoint web part class

---

**Note** If you have the choice, create ASP.NET 2.0 web parts instead of SharePoint legacy web parts; this is the approach recommended by Microsoft.

---

## HYBRID WEB PARTS

Besides ASP.NET 2.0 and SharePoint 2003 legacy web parts, there is a third flavor of web parts: hybrid web parts. Hybrid web parts, just like SharePoint legacy web parts, inherit from `Microsoft.SharePoint.WebPartPage.WebPart`. Hybrid web parts use ASP.NET 2.0 web part development techniques and capabilities combined with SharePoint capabilities. There are a couple of things you can do to make a web part hybrid:

- Use the `IPersonalizable` interface. This interface is new in .NET 2.0 and defines capabilities for the extraction of personalization state.
- Use the `[Personalizable]` attribute, which enables a particular property on a web part for personalization.
- Omit the use of XML serialization attributes.

There are a couple of drawbacks associated with creating hybrid web parts, things you can do in a normal SharePoint legacy web part but cannot do in hybrid web parts:

- Hybrid web parts cannot return tool parts via the `GetToolParts()` method. Instead, you will need to use controls that inherit from the `EditorPart` class (via the `CreateEditorPart()` method).
- Properties within a hybrid web part that are serializable will not be persisted unless you define a type converter for them.

However, some SharePoint capabilities can be leveraged within hybrid web parts. These capabilities are considered advantages of hybrid web parts:

- You can use the SharePoint 2003 web part caching framework to store items per user or per web part in memory or in SQL Server.
- You can use the SharePoint 2003 web part connection interfaces. You might want to use these interfaces when you want to create web parts that support cross-page connections or client-side communication. Web part connection interfaces are discussed in detail in Chapter 5.
- You can use the asynchronous features of web parts.

Although hybrid web parts are a somewhat esoteric topic and our guess is that you will not see them used often, it is good to know they exist. In Microsoft Office SharePoint Server 2007, we do not recommend that you create hybrid web parts anymore. The reason you might have wanted to create a hybrid web part in the past is for migration purposes; in SharePoint 2003, you could have created a hybrid web part that combined SharePoint capabilities with ASP.NET 2.0 web part development techniques. Such a hybrid web part would operate like a normal ASP.NET 2.0 web part that is easy to migrate to Microsoft Office SharePoint Server 2007.

In the final part of this section, we will take a closer look at the anatomy of the ASP.NET 2.0 web part. The most important parts of an ASP.NET 2.0 web part are as follows:

- `OnInit()`: This event handler is called immediately before the `OnInit()` method of the page that hosts the web part. This method can be used to initialize values required within the web part.
- `OnLoad()`: This event is called immediately before the `OnLoad()` method of the page that hosts the web part. This method is typically used to interact with the controls that are part of the web part.
- `CreateChildControls()`: This method can be used to add child controls to a web part and define event handlers for those child controls.
- `PreRender()`: This is the last event that occurs before the web part output is rendered to the page.
- `Render()`: This method sends the web part to its HTML writer. This method calls the following methods: `RenderBeginTag()`, `RenderContents()`, and `RenderEndTag()`.
- `RenderContents()`: This method is responsible for adding content to the web part's HTML writer.
- `Unload()`: This event occurs when the instance of the web part is discarded; at that time, the response is already sent back to the client. This is a good place to release any handles to resources that are still left open.

Listing 1-4 shows an ASP.NET 2.0 web part that incorporates the basic anatomy, as described previously.

**Listing 1-4.** *Example Web Part Incorporating Important Parts of a Web Part Anatomy*

```
using System;
using System.Runtime.InteropServices;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Serialization;

using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;
using Microsoft.SharePoint.WebPartPages;

namespace LoisAndClark.WPLibrary
{
    [Guid("c22d5295-81a3-4a23-97ed-3a0a3e8caecf")]
    public class WPLibrary : System.Web.UI.WebControls.WebParts.WebPart
    {
        public WPLibrary() : base()
        { }

        protected override void OnInit(EventArgs e)
        {
            base.OnInit(e);
        }

        protected override void OnLoad(EventArgs e)
        {
            base.OnLoad(e);
        }
    }
}
```



```

protected override void CreateChildControls()
{
    TextBox txtAddress = new TextBox();
    txtAddress.ID = "txtAddress";
    Controls.Add(txtAddress);

    Button btnSubmit = new Button();
    btnSubmit.ID = "btnSubmit";
    btnSubmit.Click += new EventHandler(btnSubmit_Click);
    Controls.Add(btnSubmit);
}

void btnSubmit_Click(object sender, EventArgs e)
{
    throw new Exception("The method or operation is not implemented.");
}

protected override void OnPreRender(EventArgs e)
{
    base.OnPreRender(e);
}

protected override void Render(HtmlTextWriter writer)
{
    base.Render(writer);
}

protected override void RenderContents(HtmlTextWriter writer)
{
    writer.Write("Hello world!");
}

protected override void OnUnload(EventArgs e)
{
    base.OnUnload(e);
}
}
}

```

This concludes our web part overview. Detailed information about creating web parts falls outside the scope of this book. If you want to learn more about creating web parts that support ASP.NET Ajax, please refer to Chapter 2. For detailed information about creating connectable web parts, please refer to Chapter 5.

## WSS 3.0 Tools: Visual Studio 2005 Extensions

When you start using Visual Studio 2005 to develop custom SharePoint solutions, one of the first things you should do is download Visual Studio 2005 extensions for Windows SharePoint Services 3.0 (VSeWSS). This is a set of tools for developing custom SharePoint 2007 solutions, supporting Windows SharePoint Services 3.0 (or any product built on top of it). The tools can be downloaded from the following location: <http://www.microsoft.com/downloads/details.aspx?familyid=19f21e5e-b715-4f0c-b959-8c6dcdbdc1057&displaylang=en>. After the download, double-click on VSeWSS.exe and follow the installation procedure. In the rest of this section, we will assume you have correctly installed Visual Studio 2005 extensions for Windows SharePoint Services 3.0.

## Project Templates

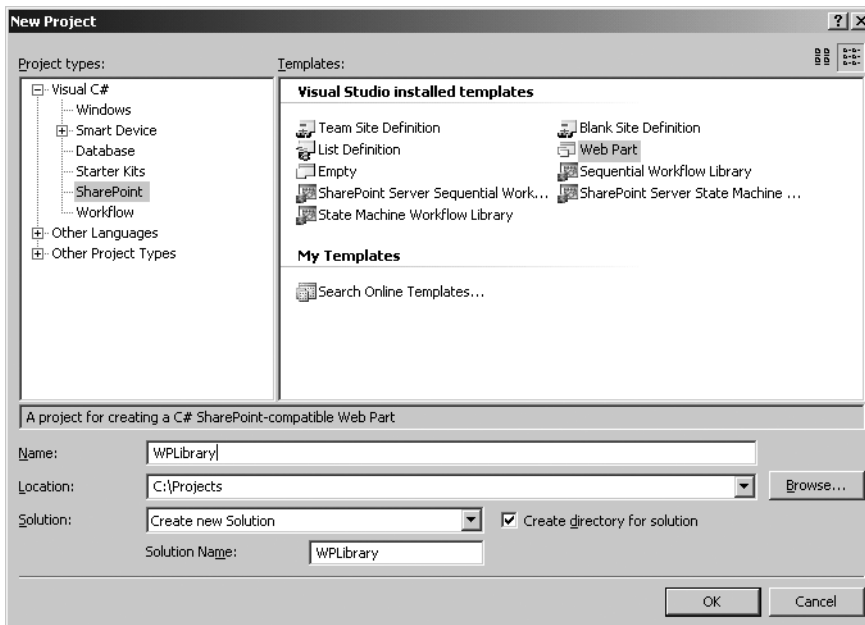
If you have installed VSeWSS and then start Visual Studio 2005 again, you will find that a new set of Visual Studio project templates is available when you create a new project. The new project templates are available in the Visual C# ► SharePoint section, as shown in Figure 1-4. The following project templates are added by Visual Studio 2005 extensions for Windows SharePoint Services 3.0:

- Team Site Definition
- List Definition
- Empty
- Blank Site Definition
- Web Part

---

**Note** Figure 1-4 contains other templates in the SharePoint project types section that are related to workflows. Those templates are not installed by Visual Studio 2005 extensions for Windows SharePoint Services 3.0, and will be discussed in detail in Chapter 4.

---



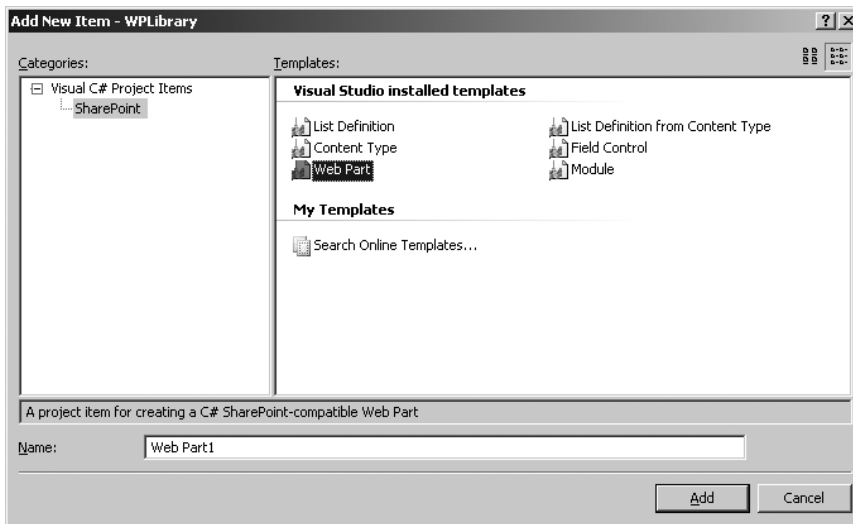
**Figure 1-4.** SharePoint Visual Studio project templates

Visual Studio 2005 extensions for Windows SharePoint Services 3.0 also contains a set of item templates that can be used within SharePoint projects for Visual Studio 2005. The following item templates are added by VSeWSS:

- List Definition
- Content Type

- Web Part
- List Definition from Content Type
- Field Control
- Module

The item templates for Visual Studio are found in the Visual C# Project Items ► SharePoint section of the Add New Item – [name of project] window. This is shown in Figure 1-5.



**Figure 1-5.** *SharePoint Visual Studio item templates*

The following section discusses the Web Part project template as well as the similarly named Web Part item template. Although the other templates are certainly interesting enough to look at, discussing them falls outside the scope of this chapter.

---

**Note** It is somewhat confusing that the project template and item template for creating web parts share the same name. In our opinion, it would have been better if the project template had been called Web Part Library.

---

## Web Part Project Template

The Web Part project template for Visual Studio 2005 extensions for Windows SharePoint Services 3.0 makes creating web part libraries easy. In this section, you will learn how to create a web part library, how to create a web part, and how to deploy and debug a web part. The following procedure explains how to create a web part library:

1. Start Visual Studio .NET 2005.
2. Choose File ► New ► Project. This opens the New Project window.
3. Select Visual C# ► SharePoint and choose the following template: Web Part.
4. In the Name text field, enter the following name: **WPLibrary**.

5. In the Location text field, enter the following location: **c:\projects**.
6. Make sure the Create directory for Solution check box is selected.
7. Click OK.

A new web part library based on the Web Part project template is created. It automatically contains references to the Microsoft.SharePoint assembly, it is strong named (see the section “Installing and Using the Web Part Library Template” in this chapter for more information about strong naming), and it contains a WPLibrary folder that contains a default web part. If some of the default settings are not acceptable to you, now is a good time to change them.

---

**Note** If you like the idea of having short assembly names and longer, descriptive namespaces, you will probably want to change the default namespace of the project. The project template adds a reference to a strong-named key file called Temporary .snk, so you will definitely need to change this and add a reference in your project to a strong-named key file (.snk file) that is used by you or your company to identify the developer of the web part library.

---

In the next procedure, we will add a new web part and have it write a “Hello!” message to the page that contains the web part.

1. Right-click the WPLibrary folder in the Solution Explorer, and choose Add ► New Item. This opens the Add New Item - [name of project] window.
2. Under Visual C# Project Items ► SharePoint, select the Web Part item template.
3. In the Name text field, enter the following name: **MyWebPart**.
4. Click Add. This adds a new MyWebPart folder that includes a class called MyWebPart.cs.
5. Open MyWebPart.cs. This is an ASP.NET 2.0 web part.
6. Add the following code to the body of the MyWebPart.cs class:

```
protected override void RenderContents(HtmlTextWriter writer)
{
    writer.Write("Hello!");
}
```

One of the nice features of the Web Part project template is the ability to package a web part within the project as a SharePoint feature. Features make it easy to deploy and activate web parts (and they can do a lot more than that).

You have fine-grained influence over the settings of a web part feature. The following procedure explains how to configure feature settings:

1. Right-click the WPLibrary project and choose Properties.
2. Click the SharePoint solution tab.

---

**Note** More information about the available settings can be found in the MSDN article “Creating a Windows SharePoint Services 3.0 Web Part Using Visual Studio 2005 Extensions” (<http://msdn2.microsoft.com/en-us/library/aa973249.aspx>) in the section “Customizing the Web Part Solution Package.”

---

At this point, you are almost ready to deploy the web parts in your web part solution. First, you must configure to which SharePoint site the web part solution needs to be deployed. This is explained in the next procedure:

1. Right-click WPLibrary and choose Properties.
2. Click the Debug tab.
3. In the Start Action section, make sure the Start browser with URL radio button is selected.
4. Enter the SharePoint site URL into the Start browser with URL text field.

---

**Note** If you fail to specify a correct SharePoint site URL, the following error message appears during deployment: “No SharePoint Site exists at the specified URL.”

---

Deploying a web part has never been easier; all you need to do is press F5 (or Debug ► Start Debugging). When you do this, a couple of things happen:

1. The web part solution is compiled.
2. The web part solution is packaged as a feature.
3. If a previous version of the web part feature exists, it is removed.
4. The web part library is added to the Global Assembly Cache (GAC).
5. The web part feature is deployed to the SharePoint site URL you specified earlier.
6. If deployment is successful, the web part feature is activated.
7. The web parts in the web part library are added to the <SafeControls> list in the web.config file of the SharePoint web application that contains the web part.
8. The web part gallery is modified so that the site can use the web parts.
9. Internet Information Services (IIS) is restarted.
10. If debugging is enabled, Visual Studio 2005 automatically tries to attach to the w3wp process hosting the web part.

---

**Note** The bin\debug folder of a web part library created using Visual Studio 2005 contains a setup.bat file that can be customized to deploy the web parts created in the web part library. You can find more information about web part deployment in the section “Deploying a Web Part,” later in this chapter.

---

Now, you should go ahead and add the web part to a web part page. The following procedure explains how to do this:

1. Navigate to the SharePoint site URL you specified earlier.
2. Choose Site Actions ► Edit Page. This opens the page in Edit Mode.
3. Choose a web part zone and click the Add a Web Part link. This opens the Add Web Parts — Web Page Dialog window.
4. Locate the All Web Parts ► Miscellaneous section and select the MyWebPart Web Part check box.
5. Click the Add button.

This adds the MyWebPart web part to the page. If you want to go ahead and debug this web part, follow these steps:

1. Open a command prompt and type **inetmgr**. This opens Internet Information Services (IIS) Manager.
2. Expand the [computer name] (local computer) node.
3. Expand the Web Sites node.
4. Locate the SharePoint web site that contains your web part, right-click it, and choose Properties. This opens the [web site name] Properties window.
5. Click the Home Directory tab.
6. Copy the value of the Local Path text field.
7. Open Windows Explorer and navigate to the local path you copied in the previous step. This opens the root folder of the SharePoint web application that contains your web part.
8. Set a break point to a line of code in the web part.
9. Press F5.

After deployment, Visual Studio 2005 attaches automatically to the process hosting your web part, and break mode is entered automatically. This is shown in Figure 1-6.

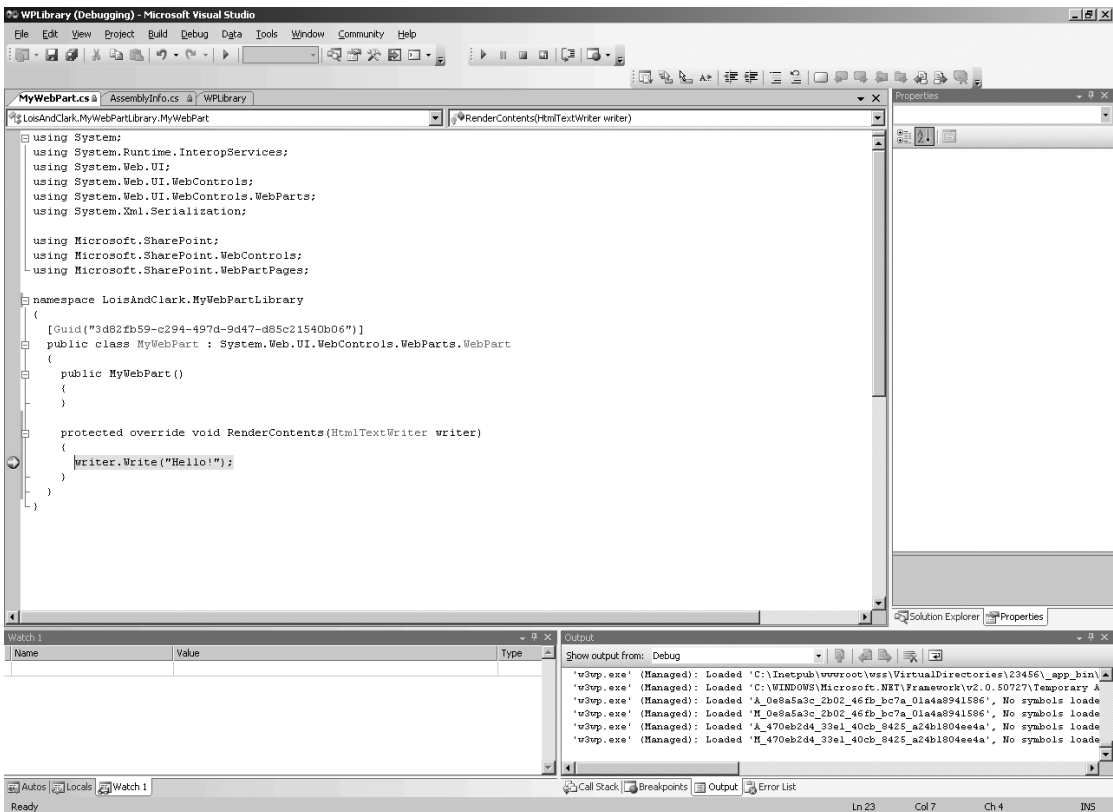


Figure 1-6. Using the web part template to debug a web part

## DEBUGGING WEB PARTS

Whenever an error occurs in a web part, Microsoft Office SharePoint Server 2007 shows a generic error message. If you want to see a detailed error message, you should open the web.config file of the SharePoint web application that contains the web part, locate the <SafeMode> node, and set the CallStack attribute to true. If you want to be able to see more than a generic server error message, you should locate the <customErrors> element and set the mode attribute to Off.

You do not need to set the debug property of the <compilation> element to true to be able to debug a web part. This setting tells ASP.NET to generate symbol tables (.pdb files) when compiling pages. Since your web part code is contained in an assembly, you can debug the necessary .pdb files by building a Debug version of the web part library. When the debug symbols are present, you can debug web parts by attaching to the process (an instance of w3wp.exe).

However, if you want to be able to debug web parts by letting Visual Studio 2005 auto-attach to the process, you do need to set the debug attribute of the <compilation> attribute to true. In that case, you also need to raise the trust level for the SharePoint web application containing your web part to WSS\_Medium or greater.

Be careful: setting the debug attribute of the <compilation> attribute to true has an undesirable side effect. Every time you run the web part library project, a .bak copy of your web application's web.config file is created.

The blog post "Debugging web parts – a full explanation of the requirements," written by Maurice Prather, discusses the previously discussed techniques in detail. The blog post can be found at the following location: <http://www.bluedoglimited.com/SharePointThoughts/ViewPost.aspx?ID=60>.

Whenever you are auto-attaching to a process in Visual Studio 2005, the web server is restarted in order to recycle the w3wp.exe process hosting your web part. This is overkill. Instead, it is much faster if you just restart the application pool that hosts the SharePoint web application containing your web part and then manually attach to the process hosting your web part. You can restart the application pool via iisapp.vbs (located in the [drive letter]:\windows\system32 folder) using the following command:

```
iisapp.vbs /a "[application pool name]" /r
```

You can make attaching to this process even easier by installing the Debugger feature for SharePoint, created by Jonathan Dibble. This feature adds a new menu item to the Site Actions menu of a SharePoint site and is described in detail in the blog post "Debugger 'Feature' for SharePoint" at <http://blogs.msdn.com/sharepoint/archive/2007/04/10/debugger-feature-for-sharepoint.aspx>. The feature can be downloaded from the SharePoint 2007 Features site at CodePlex (<http://www.codeplex.com/features/Release/ProjectReleases.aspx?ReleaseId=2502>). This list of free SharePoint features is maintained by Scot Hillier.

## SharePoint Solution Generator

You have seen that Visual Studio 2005 extensions for Windows SharePoint Services 3.0 contains a set of project and item templates. The final part of the Visual Studio 2005 extensions is the SharePoint Solution Generator. This is a stand-alone tool that can be started by choosing Start ► All Programs ► SharePoint Solution Generator. SharePoint Solution Generator is a convenient tool that allows you to generate either site definitions or list definitions in the form of Visual Studio 2005 projects (.csproj files) based on existing SharePoint sites and lists.

---

**Caution** The current version of the tool (version 1.0) breaks if the default IIS web site is not extended with SharePoint functionality. You can extend the default IIS web site via SharePoint Central Administration ► Application Management ► Create or extend Web Application. If the default IIS web site is not extended with SharePoint functionality, you will get an unhandled exception once you start generating a Site or List definition.

---

## ASP.NET 2.0 Server Controls

In this section, we will show you a couple of the ASP.NET 2.0 server controls used within a SharePoint web part. ASP.NET 2.0 server controls contain great functionality for database access, calendars, text boxes, drop-down lists, and a lot of other common composite web functionality. Controls are essential to the ASP.NET programming model. In ASP.NET 2.0, there are nearly 60 new server controls.

### Configuring Web Parts

If you are building a generic web part, you probably will need to be able to configure it on a per-web part basis. You can use web part properties and the Web Part Editor tool pane to configure web parts, as you will see in this section. Such configuration information is stored in the SharePoint content database.

To demonstrate web part configuration via properties, we will create a web part that uploads a local file to a SharePoint document library. File upload in ASP.NET 1.x was possible, although you did have to jump through some hoops to get everything working. For example, you had to add `enctype="multipart/form-data"` to the page's `<form>` element. The new ASP.NET 2.0 FileUpload server control makes the process of uploading files to the hosting server as simple as possible.

The FileUpload control displays a text box and a Browse button that allow users to select a file to upload to the server. The user specifies the file to upload by entering the fully qualified path to the file on the local computer (for example, `C:\Temp\Test.txt`) in the text box of the control. The user can also select the file by clicking the Browse button and then locating it in the Choose File dialog box. You need to hook up an event handler to a Submit button, which calls the `SaveAs()` method of the FileUpload control. Via this method, you can specify the location where the file will be saved. The file will not be uploaded to the server until the user clicks the Submit button.

To build this web part, we need three configurable properties to store the following information:

- Site collection URL
- Site URL
- Name of the document library that is going to hold the uploaded file

If you want to create a configurable property, you first need to add the `[Personalizable]` attribute to a property to indicate that the property supports personalization. You also need to specify the personalization scope that associates the scope of a web part property to the state a web part page is running in. This determines which configuration data is retrieved from the SharePoint configuration database. You can set the personalization scope to either `Shared` or `User`.

Setting the personalization scope of a web part to `Shared` ensures that the data associated with a page running in `Shared` mode is retrieved. In this case, the configuration information applies to all users viewing the web part.

You can also set the personalization scope of a web part to `User` to make sure that the data associated to a page running in `User` mode is retrieved. Such configuration information indicates that configuration information is retrieved for the currently executing user.



In this example, we are not interested in user-specific data; we are interested in application-specific data. As a result, we will set the personalization scope to Shared. You will also need to set the [WebBrowsable] attribute to true to make it appear in the web part tool pane. In addition, you can use the [WebDisplayName] and [WebDescription] attributes to define a friendly name and description for the web part property. This information is shown in the web part tool pane. The following code fragment shows a web part property containing the URL of a site collection that is personalizable, stores configuration on an application-wide basis, and defines a friendly name and description:

```
private string _strSiteCollectionUrl;
[Personalizable(PersonalizationScope.Shared),
WebBrowsable(true),
WebDisplayName("Site Collection URL"),
WebDescription("Enter the URL of the site collection that contains the
list a document is uploaded to")]
public string SiteCollectionUrl
{
    get { return _strSiteCollectionUrl; }
    set { _strSiteCollectionUrl = value; }
}
```

The web part in this section opens a site collection containing the document library that will hold the uploaded file. Then, it opens the site that contains this document library, and finally, it opens the document library itself. In this example, we are saving uploaded files to the root folder of the document library by using the Add() method of the RootFolder object's Files collection. By passing the input stream of the FileUpload control to the Add() method, you can save a file to a document library. The next code fragment shows how to save a file uploaded via the FileUpload control in a document library, overwriting files that have the same name:

```
using (SPSite objSite = new SPSite(SiteCollectionUrl))
{
    using (SPWeb objWeb = objSite.OpenWeb(SiteUrl))
    {
        SPList objList = objWeb.Lists[ListName];
        if (_objFileUpload.HasFile)
        {
            objList.RootFolder.Files.Add(_objFileUpload.FileName, ➤
            _objFileUpload.PostedFile.InputStream, true);
        }
    }
}
```

The complete code for a configurable web part that uses the FileUpload control to upload a file to a SharePoint document library is shown in Listing 1-5.

#### **Listing 1-5.** *The FileUpload Web Part*

```
using System;
using System.Runtime.InteropServices;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Serialization;

using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;
using Microsoft.SharePoint.WebPartPages;
```

```

namespace LoisAndClark.AspNetExample
{
    [Guid("288802c4-4dfe-45b6-bb28-49dda89ec225")]
    public class NormalPart : System.Web.UI.WebControls.WebParts.WebPart
    {
        FileUpload _objFileUpload = new FileUpload();

        protected override void CreateChildControls()
        {
            Controls.Add(_objFileUpload);

            Button btnUpload = new Button();
            btnUpload.Text = "Save File";
            this.Load += new System.EventHandler(btnUpload_Click);
            Controls.Add(btnUpload);
        }

        private void btnUpload_Click(object sender, EventArgs e)
        {
            using (SPSite objSite = new SPSite(SiteCollectionUrl))
            {
                using (SPWeb objWeb = objSite.OpenWeb(SiteUrl))
                {
                    SPList objList = objWeb.Lists[ListName];
                    if (_objFileUpload.HasFile)
                    {
                        objList.RootFolder.Files.Add(_objFileUpload.FileName,
                            _objFileUpload.PostedFile.InputStream, true);
                    }
                }
            }
        }

        private string _strSiteCollectionUrl;
        [Personalizable(PersonalizationScope.Shared), WebBrowsable(true),
        WebDisplayName("Site Collection URL"),
        WebDescription("Enter the URL of the site collection that contains
        the list a document is uploaded to")]
        public string SiteCollectionUrl
        {
            get { return _strSiteCollectionUrl; }
            set { _strSiteCollectionUrl = value; }
        }

        private string _strSiteUrl;
        [Personalizable(PersonalizationScope.Shared),
        WebBrowsable(true), WebDisplayName("Site Collection URL"),
        WebDescription("Enter the URL of the site that contains
        the list a document is uploaded to")]
        public string SiteUrl
        {
            get { return _strSiteUrl; }
            set { _strSiteUrl = value; }
        }

        private string _strListName;
        [Personalizable(PersonalizationScope.Shared),

```

```

WebBrowsable(true),
WebDisplayName("List name"), WebDescription("Enter the name of the list that
will contain the uploaded document")]
public string ListName
{
    get { return _strListName; }
    set { _strListName = value; }
}
}
}

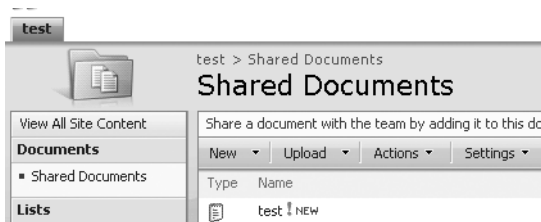
```

Figure 1-7 shows the FileUpload web part (called NormalPart) in action. First, the web part allows you to choose a local file (test.txt) that you want to upload to a SharePoint document library.



**Figure 1-7.** Choosing a local file to upload

Figure 1-8 shows the end result. The local file test.txt is uploaded to a document library called Shared Documents.



**Figure 1-8.** The local file is successfully uploaded to the Shared Documents document library.

## Using Resources in Web Parts

Most web parts use external resources such as images or JavaScript libraries. Such resources can be deployed to the `_layouts` virtual directory of a SharePoint web application, which is a suitable location because every user has sufficient rights to access it without being able to modify the resources located there. However, since the advent of ASP.NET 2.0, there is another possibility: web resources. In this section, we will use the `BulletedList` server control to demonstrate the use of web resources.

---

**Note** The images subfolder of the `_layouts` folder is an excellent place to store custom resources because every user has sufficient rights to read resources in it. The images folder can be found at the following location: [drive letter]:\Program Files\Common Files\Microsoft Shared\web server extensions\12\TEMPLATE\IMAGES\.

---

The `BulletedList` control, as the name says, displays a bulleted list of items in an ordered (`<ol>` HTML element) or unordered (`<ul>` HTML element) fashion. The `BulletedList` control has extra

features, such as the option to choose bullet style, data binding, and support for custom images. The bullet style lets you choose the style of the element that precedes the item. For example, you can choose to use numbers, squares, or circles. Child items can be rendered as plain text, hyperlinks, or buttons. The web part example in this section shows a bulleted list made with the `BulletedList` control that uses a custom image.

Web resources allow you to include external resources such as images and JavaScript libraries in assemblies. This allows you to include such resources in the web part assembly, if you want. For reasons of better component design, we have decided to create a new assembly that contains web resources.

---

**Note** This is also known as the Common-Closure principle, which requires that the content of an assembly should be closed against the same kinds of code changes. If you want to learn more about component design, we advise you to read the book *Agile Software Development* by Robert C. Martin.

---

If you want to create web resources, you need to add them to an assembly and mark them as Embedded Resources using Visual Studio 2005 by setting the resource's Build Action property to Embedded Resource. After doing that, you need to add a `[WebResource]` attribute for each embedded resource in the assembly to enable the resource to be used as a web resource. The constructor for the `[WebResource]` attribute accepts two parameters: web resource and content type. The web resource consists of three parts:

- *Assembly namespace*: For example, `LoisAndClark.WebResources`
- *Folder structure*: For example, `folder1.childfolder2.grandchildfolder3`
- *Resource name*: For example, `favicon.gif`

The following code fragment shows how to mark `favicon.gif` as a web resource:

```
[assembly: WebResourceAttribute(  
    "LoisAndClark.WebResources.gfx.favicon.gif",  
    "image/gif"  
)]
```

You will also need to create a dummy class, which we will call `Placeholder.cs`. This class does not actually need to do anything, but it is used by clients to obtain a reference to the resource assembly when retrieving resource URLs. This will be shown later in this section. Here is the code fragment for the dummy class:

```
public class Placeholder  
{}
```

In the next procedure, you will learn how to create a new assembly that contains a custom image. The custom image is used by a `BulletedList` control to display a list in a web part called `NormalPart`:

1. Open web part library project. If you are not sure how to create a web part library project, please refer back to the section "WSS 3.0 Tools: Visual Studio 2005 Extensions."
2. Choose File ► Add ► New Project. This opens the Add New Project wizard.
3. In the Project types section, click Visual C#.
4. In the Templates section, choose Class Library.
5. Choose the following name: `WebResources`.
6. Choose the following location: `c:\projects\wpliblibrary`.

7. Click OK.
8. Rename Class1.cs to Placeholder.cs.
9. Rename the namespace from WebResources to LoisAndClark.WebResources.
10. Right-click the References folder and choose Add Reference. This opens the Add Reference window.
11. In the .NET tab, locate System.Web and click OK.
12. Right-click the WebResource project and choose Properties. This opens the WebResources page.
13. Click the Application tab and add the following value in the Default namespace textbox: **LoisAndClark.WebResources**.
14. Click the Signing tab and select the Sign the assembly check box.
15. In the Choose a Strong-Name Key file drop-down list, choose <New...>. This opens the Create Strong-Name Key window.
16. In the Key file name text box, add the following value: **WebResourcesKey**.
17. Do not select the Protect my key file with a password check box.
18. Click OK.
19. Right-click the WebResources project and choose Add ► New Folder.
20. Enter the following folder name: **gfx**.
21. Right-click the gfx folder and choose Add ► Existing Item. This opens the Add Existing Item - WebResources window.

---

**Note** In this example, we are using a custom image called favicon.gif. If you do not want to break the code that is discussed later, you should note that casing is important.

---

22. Click favicon.gif. In the Properties window, set the Build Action property to Embedded Resource.
23. In the WebResources project, expand the Properties node and double-click AssemblyInfo.cs.
24. Add the following statement to the top of AssemblyInfo.cs:

using System.Web.UI;

25. At the bottom of AssemblyInfo.cs, add the following attribute:

```
[assembly: WebResourceAttribute(
    "LoisAndClark.WebResources.gfx.favicon.gif",
    "image/gif"
)]
```

---

**Note** The first part of a WebResource constructor accepts the name of the resource, consisting of the namespace of the assembly, the folder structure, and the name of the resource itself. Remember that casing is important here! The second part of the constructor accepts the type of the resource. In this case, `image/gif` is used to indicate that the web resource is an image. The most common other option is to include a JavaScript library as a web resource. If you want to add a JavaScript library instead of an image, use `text/javascript` instead of `image/gif`.

---

At this point, you have created a new assembly called `WebResources` that contains the custom `favicon.gif` image. Next, we will show you how to build a web part that displays a custom image that is deployed as a web resource in a bulleted list.

---

**Note** When resource assemblies are compiled in debug mode, caching is not enabled. When resource assemblies are compiled in release mode, caching is enabled.

---

You can use the `GetWebResourceUrl()` method of the `Page.ClientScript` object to obtain a URL reference to a web resource. The first argument of this method accepts the type of the server resource, which can be obtained by determining the type of the dummy class called `Placeholder.cs`, like so:

```
typeof(LoisAndClark.WebResources.PlaceHolder)
```

The second argument is the name of the server resource, consisting of the namespace, folder structure, and name of the resource. In this example, the fully qualified name of the resource is

```
LoisAndClark.WebResources.gfx.favicon.gif
```

The next code fragment shows the complete code for obtaining a URL reference to a web resource:

```
Page.ClientScript.GetWebResourceUrl(
    typeof(LoisAndClark.WebResources.PlaceHolder), strResource);
```

The URL that is retrieved will look like this:

```
/WebResource.axd?d=[Assembly key]&t=[last write time of resource assembly]
```

The next procedure explains how to create the `NormalPart` web part that displays a bulleted list using a custom image that is available as a web resource:

1. Switch to your web part library. Ours is called `WPLibrary`.
2. Right-click the `References` node and choose `Add Reference`. This opens the `Add Reference` window.
3. Click the `Projects` tab. Select `WebResources` and click `OK`.
4. Open the web part class file. Ours is called `NormalPart`.
5. Add the code from Listing 1-6.

**Listing 1-6.** *Consuming Resources Within a Web Part*

```
using System;
using System.Runtime.InteropServices;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Serialization;
using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;
using Microsoft.SharePoint.WebPartPages;
using Microsoft.SharePoint.Administration;
using Microsoft.SharePoint.Utilities;
using System.Net;
using System.Net.Mail;
```

```

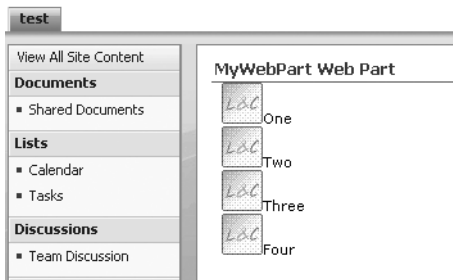
namespace LoisAndClark.AspNetExample
{
    [Guid("288802c4-4dfe-45b6-bb28-49dda89ec225")]
    public class NormalPart : System.Web.UI.WebControls.WebParts.WebPart
    {
        protected override void CreateChildControls()
        {
            string strResource = "LoisAndClark.WebResources.gfx.favicon.gif";
            string strUrl = ➡
            Page.ClientScript.GetWebResourceUrl( ➡
            typeof(LoisAndClark.WebResources.PlaceHolder), strResource);

            BulletedList objBullist = new BulletedList();
            objBullist.BulletStyle = BulletStyle.CustomImage;
            objBullist.BulletImageUrl = strUrl;
            objBullist.Items.Add("One");
            objBullist.Items.Add("Two");
            objBullist.Items.Add("Three");
            objBullist.Items.Add("Four");

            Controls.Add(objBullist);
        }
    }
}

```

Now, you have created all the code you need. The last step you need to perform is to deploy the WebResources assembly to the Global Assembly Cache (GAC) to make sure the web resources are available to the web part. One way to do this is to open [drive letter]:\Windows\Assembly. Drag and drop WebResources.dll into the Global Assembly Cache. The resulting BulletedList control is shown in Figure 1-9.



**Figure 1-9.** *The BulletedList control in a web part*

Web resources are great for packaging resources in assemblies. This allows you to tie specific versions of an assembly to a specific set of resources. However, there are some drawbacks associated to the use of web resources:

- You cannot embed every type of resource in an assembly. For instance, you cannot embed .asmx, .ascx, and web.config files as web resources.
- It is hard, if not impossible, to debug embedded resources.

Alternatively, you could store resources in a separate virtual directory.

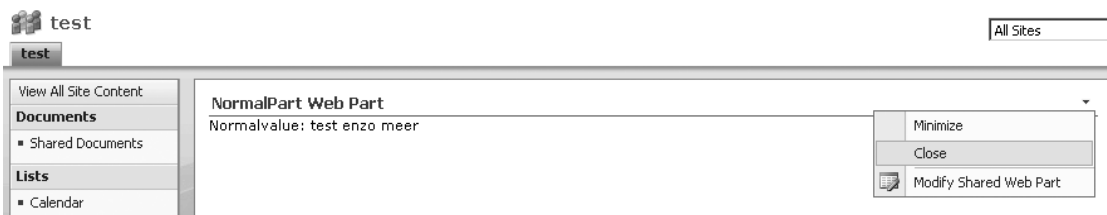
---

**Note** It is possible to deploy resources using a solution (.wsp) file. Creating solution files is discussed in detail in the section “Deploying a Web Part,” later in this chapter. You can use the `Location` attribute of the `<ClassResource>` element in the web part manifest file to specify resource files. Do not use the `FileName` attribute; this attribute is still present but it is only used in SharePoint 2003.

---

## Web Part Menu Verbs

Every web part on a SharePoint page has a web part menu that contains standard options such as the ability to minimize or close the web part. An example of a standard web part menu is shown in Figure 1-10.



**Figure 1-10.** *The standard web part menu*

In this section, you will learn how to create custom web part menu verbs that are added to the verbs menu in a web part’s header. To start with, it is good to realize that there are three kinds of web part menu verbs:

- *Client-side verbs.* These verbs perform some kind of action on the client side.
- *Server-side verbs.* These verbs perform some kind of action on the server side.
- *Both.* These verbs perform actions on both the client and server side.

You can create a client-side verb by creating a new instance of the `WebPartVerb` class. The first argument that needs to be passed to the `WebPartVerb` constructor for a client-side verb is the verb’s id. This needs to be unique. The second argument refers to the client-click event handler. This typically refers to a piece of JavaScript that needs to be executed once an end user clicks on a web part menu verb. The following code fragment shows how to create a client-side verb:

```
WebPartVerb objFirst = new WebPartVerb("FirstVerbId", ➤
"javascript:alert('Hello from verb!');");
objFirst.Text = "first verb text";
objFirst.Description = "first verb description";
objFirst.ImageUrl = "_layouts/images/loisandclark/favicon.gif";
```

If you want to create a server-side verb, you need to instantiate a new `WebPartVerb` class, pass it a unique verb id, and specify a server-side event handler to be executed once an end user clicks on the verb. The next code fragment shows how to create a server-side verb:

```
WebPartVerb objSecond = new WebPartVerb("SecondVerbId",
new WebPartEventHandler(SecondVerbHandler));
objSecond.Text = "second verb text";
objSecond.Description = "second verb description";
```



The final web part menu verb type, the kind that performs both client-side and server-side actions, is created by passing a unique verb id, a server-side event handler, and a client-side event handler.

You can add custom web part verbs by overriding the `Verbs` property of a web part and adding custom verbs to it. Listing 1-7 shows how to add custom verbs to the web part menu and define an event handler that handles a server-side verb click event:

**Listing 1-7.** *Adding Custom Verbs to a Web Part Menu*

```
using System;
using System.Runtime.InteropServices;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Serialization;
using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;
using Microsoft.SharePoint.WebPartPages;
using Microsoft.SharePoint.Administration;
using Microsoft.SharePoint.Utilities;
using System.Net;
using System.Net.Mail;

namespace LoisAndClark.AspNetExample
{
    [Guid("288802c4-4dfe-45b6-bb28-49dda89ec225")]
    public class NormalPart : System.Web.UI.WebControls.WebParts.WebPart
    {
        public override WebPartVerbCollection Verbs
        {
            get
            {
                WebPartVerb objFirst = new WebPartVerb("FirstVerbId", ➤
                "javascript:alert('Hello from verb!');");
                objFirst.Text = "first verb text";
                objFirst.Description = "first verb description";
                objFirst.ImageUrl = "_layouts/images/loisandclark/favicon.gif";

                WebPartVerb objSecond = new WebPartVerb("SecondVerbId", ➤
                new WebPartEventHandler(SecondVerbHandler));
                objSecond.Text = "second verb text";
                objSecond.Description = "second verb description";

                WebPartVerb[] objVerbs = new WebPartVerb[] {objFirst, objSecond};
                WebPartVerbCollection objVerbCollection = ➤
                new WebPartVerbCollection(base.Verbs, objVerbs);

                return objVerbCollection;
            }
        }

        protected void SecondVerbHandler(object sender, WebPartEventArgs args)
        {
            //Do something...
        }
    }
}
```

This code adds two web part verbs to the web part menu: a client-side verb and a server-side verb. Both verbs are shown in Figure 1-11.



**Figure 1-11.** A web part menu containing two custom verbs

Figure 1-12 shows what happens when you click on the client-side verb. Some JavaScript will be executed that displays an alert box.



**Figure 1-12.** The client-side event handler for the client-side verb in action

In this section, you have seen that adding verbs to a web part menu enhances the user interface experience and is quite easy to accomplish.

## Creating Editor Parts

If you have sufficient permissions, you are allowed to configure web parts. When you start configuring web parts via the user interface, a web part tool pane opens. Any configuration information stored in the pane is persisted in the SharePoint content database. In the section “Configuring Web Parts,” earlier in this chapter, you learned how to create properties that can be configured using the web part tool pane. Figure 1-13 shows an example of a web part tool pane.

Out of the box, SharePoint offers different editors for different types of data. For instance, a string property is represented by the text box editor, while a Boolean property is represented by check boxes. In this section, you will learn how to create custom editor parts, which will be shown in the web part tool pane when you are configuring a web part.

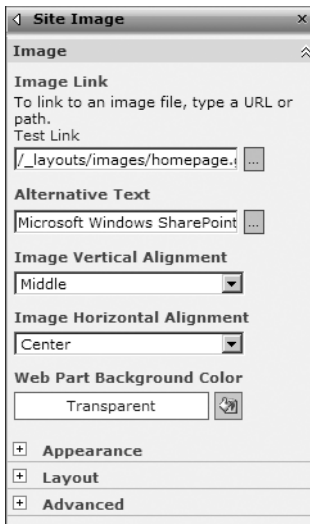
All editor parts inherit from the `EditorPart` base class, so that is the first step you need to take when creating a custom editor part. Editor parts look a lot like normal web parts. For instance, if you want to add child controls to an editor part, you need to override its `CreateChildControls()` method. The following code fragment adds a text area consisting of five lines to an editor part:

```
protected override void CreateChildControls()
{
    _txtNormalBox = new TextBox();
    _txtNormalBox.ID = "txtNormalBox";
    _txtNormalBox.Text = "[Custom editor part]";
    _txtNormalBox.TextMode = TextBoxMode.Multiline;
    _txtNormalBox.Rows = 5;
    Controls.Add(_txtNormalBox);
}
```

You will also need to override the `ApplyChanges()` method of an `EditorPart` instance. This method is responsible for mapping values in the editor part to corresponding properties in the associated web part. The following code fragment shows how the `ApplyChanges()` method sets a property in an instance of the `NormalPart` web part called `NormalValue`:

```
public override bool ApplyChanges()
{
    NormalPart objNormal = (NormalPart)WebPartToEdit;
    objNormal.NormalValue = _txtNormalBox.Text;

    return true;
}
```



**Figure 1-13.** *The default web part tool pane*

---

**Note** This code performs a downcast, which is considered a bad practice in object-oriented programming because it tightly couples the editor part and the web part. It would have been better to define an interface that is implemented by every web part that can be used in conjunction with the editor part.

---

The final piece of an editor part that must be implemented is the `SyncChanges()` method. This method is the opposite of the `ApplyChanges()` method; it retrieves property values from a web part and stores them in the editor part. The following code fragment shows an implementation for the

SyncChanges() method that retrieves the NormalValue property of a NormalPart web part and stores it in the text area of the web part editor tool pane:

```
public override void SyncChanges()
{
    // Make sure the text area child control is created.
    EnsureChildControls();

    NormalPart objNormal = (NormalPart) WebPartToEdit;
    _txtNormalBox.Text = objNormal.NormalValue;
}
```

So far, you have seen how to create an editor part. Before a custom editor part is shown in the web part editor tool pane, you need to implement support for this in your web part. In this example, we are creating a web part called NormalPart. First, we need to add one or more properties that are read from and written to within the editor part. The custom editor part in this section expects the presence of a NormalValue property in a web part, so that is the first thing that needs to be implemented. If you need more information on how to do this, please refer back to the section “Configuring Web Parts.” The following code fragment shows how to implement the NormalValue property:

```
private string _strNormalValue = String.Empty;
[Personalizable(PersonalizationScope.Shared), WebBrowsable(false), ➤
WebDisplayName("Normal value"), WebDescription("Normal value description")]
public string NormalValue
{
    get { return _strNormalValue; }
    set { _strNormalValue = value; }
}
```

After that, you need to override the CreateEditorParts() method. This method is responsible for returning a collection of custom editor part controls that are shown in the web part editor tool pane when a web part is in edit mode. Basically, this method is used to create a new instance of one or more custom editor parts and add those to the editor part collection. This collection is used when the web part tool pane is rendered. The following code fragment shows how to add the NormalEditorPart editor part to the collection of customer editor parts that is shown when the NormalPart web part is being edited:

```
public override EditorPartCollection CreateEditorParts()
{
    NormalEditorPart objEditor = new NormalEditorPart();
    objEditor.ID = ID + "normalEditor1";
    objEditor.Title = "Normal Editor title";
    objEditor.ToolTip = "Normal Editor tooltip";
    objEditor.TabIndex = 100;
    objEditor.GroupingText = "Normal editor grouping text";

    ArrayList objEditorParts = new ArrayList();
    objEditorParts.Add(objEditor);

    EditorPartCollection objEditorPartsCollection = new
    EditorPartCollection(objEditorParts);

    return objEditorPartsCollection;
}
```

Listing 1-8 shows the complete code for a custom editor part called `NormalEditorPart` that is being used in a web part called `NormalPart`:

**Listing 1-8. Creating an Editor Part**

```
using System;
using System.Runtime.InteropServices;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Serialization;

namespace LoisAndClark.AspNetExample
{
    public class NormalEditorPart : EditorPart
    {
        TextBox _txtNormalBox;

        protected override void CreateChildControls()
        {
            _txtNormalBox = new TextBox();
            _txtNormalBox.ID = "txtNormalBox";
            _txtNormalBox.Text = "[Custom editor part]";
            _txtNormalBox.TextMode = TextBoxMode.MultiLine;
            _txtNormalBox.Rows = 5;
            Controls.Add(_txtNormalBox);
        }

        public override bool ApplyChanges()
        {
            NormalPart objNormal = (NormalPart)WebPartToEdit;
            objNormal.NormalValue = _txtNormalBox.Text;

            return true;
        }

        public override void SyncChanges()
        {
            EnsureChildControls();
            NormalPart objNormal = (NormalPart)WebPartToEdit;
            _txtNormalBox.Text = objNormal.NormalValue;
        }
    }
}
```

Listing 1-9 contains the implementation of the `NormalPart` web part. This web part displays the custom `NormalEditorPart` web part in edit mode:

**Listing 1-9. Creating a Web Part Displaying a Custom Editor Part**

```
using System;
using System.Collections;
using System.Runtime.InteropServices;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Serialization;
```

```

using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;
using Microsoft.SharePoint.WebPartPages;
using Microsoft.SharePoint.Administration;
using Microsoft.SharePoint.Utilities;

using System.Net;
using System.Net.Mail;

namespace LoisAndClark.AspNetExample
{
    [Guid("288802c4-4dfe-45b6-bb28-49dda89ec225")]
    public class NormalPart : System.Web.UI.WebControls.WebParts.WebPart
    {
        protected override void RenderContents(HtmlTextWriter writer)
        {
            writer.Write("Normalvalue: " + NormalValue);
        }

        public override EditorPartCollection CreateEditorParts()
        {
            NormalEditorPart objEditor = new NormalEditorPart();
            objEditor.ID = ID + "normalEditor1";
            objEditor.Title = "Normal Editor title";
            objEditor.ToolTip = "Normal Editor tooltip";
            objEditor.TabIndex = 100;
            objEditor.GroupingText = "Normal editor grouping text";

            ArrayList objEditorParts = new ArrayList();
            objEditorParts.Add(objEditor);

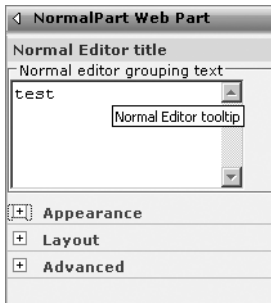
            EditorPartCollection objEditorPartsCollection = ➡
            new EditorPartCollection(objEditorParts);

            return objEditorPartsCollection;
        }

        private string _strNormalValue = String.Empty;
        [Personalizable(PersonalizationScope.Shared), ➡
        WebBrowsable(false), ➡
        WebDisplayName("Normal value"), ➡
        WebDescription("Normal value description")]
        public string NormalValue
        {
            get { return _strNormalValue; }
            set { _strNormalValue = value; }
        }
    }
}

```

Figure 1-14 shows the end result of our efforts: the editor tool pane when the web part is in edit mode. As you can see, the custom text area that is created in our custom editor part is displayed in the tool pane.



**Figure 1-14.** Adding a custom control to the web part editor tool pane

## Deploying a Web Part

If you want to deploy a web part, you can use SharePoint solution files. Basically, SharePoint solution files are cabinet files (.cab) with a different extension, namely .wsp. When you use Visual Studio 2005 extensions for Windows SharePoint Services 3.0 to create a web part, as was discussed in the section “WSS 3.0 Tools: Visual Studio 2005 Extensions,” a web part solution file is created automatically for you. In this section, you will learn how to create a SharePoint solution file that allows you to deploy a web part to other SharePoint servers and/or SharePoint web applications.

---

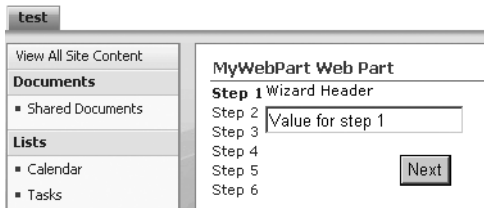
**Note** WSP files (SharePoint solution files) can be used to deploy site definitions, site features, template files, assemblies, code access security policy files, and web parts.

---

To demonstrate web part deployment, we will create a new web part called `MyWebPart.cs` that uses the ASP.NET 2.0 Wizard control to implement a wizard. The Wizard server control enables you to build a sequence of steps that are displayed to the end user. You could use the Wizard control to either display or gather information in small steps. The Wizard control lets you define a group of views in which only one view at a time is active and is rendered to the client. Each view consists of four zones: sidebar, header, content, and navigation. The optional sidebar part contains an overview of all the steps in the wizard. The header contains the header information. The content part contains whichever control or controls you like. The navigation part consists of buttons to navigate through the steps in the wizard.

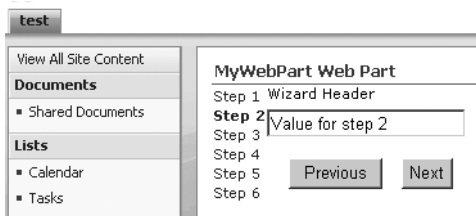
When you are constructing a step-by-step process that includes logic for every step taken, use the Wizard control to manage the entire process. The Wizard control’s navigation buttons fire server-side events whenever the user clicks one of the buttons, helping to navigate to other wizard views on the same page. Navigation can be linear and nonlinear; in other words, you can jump from one view to another or navigate randomly to whichever view you like. All the controls in a wizard view are part of the page, so you can access them in code using their control IDs.

In this example, we have defined six different steps. Each step is a `WizardStep` control and contains a text box. The state of the text boxes in the wizard is maintained automatically. The order in which the steps are defined is completely based upon the order in which they are added to the wizard via the `Add()` method of the Wizard object’s `WizardSteps` property. The `WizardSteps` property contains a collection of `WizardSteps`. Changing this order changes the order in which the end user sees them. Figure 1-15 shows the first step.



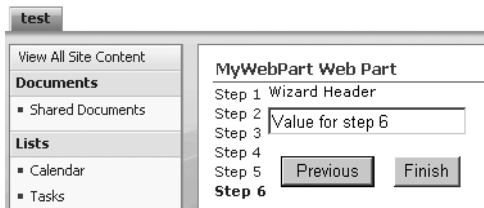
**Figure 1-15.** Step 1 of the Wizard control in a web part

The first step, the start step, always has one button called Next. The following steps will have two buttons, as seen in Figure 1-16.



**Figure 1-16.** Step 2 of the Wizard control in a web part

There are six steps in this example. The final step, step 6, has a Previous and a Finish button, as you can see in Figure 1-17.



**Figure 1-17.** The final step of the Wizard control in a web part

The MyWebPart web part creates a very simple wizard; the Wizard control itself has a lot more options that we will not cover in this chapter. The code for the web part using the Wizard control is shown in Listing 1-10:

**Listing 1-10.** *The Wizard Web Part*

```
using System;
using System.Runtime.InteropServices;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Serialization;
```



```

using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;
using Microsoft.SharePoint.WebPartPages;

namespace LoisAndClark.MyWebPartLibrary
{
    [Guid("3d82fb59-c294-497d-9d47-d85c21540b06")]
    public class MyWebPart : System.Web.UI.WebControls.WebParts.WebPart
    {
        public MyWebPart()
        {
        }

        protected override void CreateChildControls()
        {
            Wizard objWizard = new Wizard();
            objWizard.HeaderText = "Wizard Header";

            for (int i = 1; i <= 6; i++)
            {
                WizardStepBase objStep = new WizardStep();
                objStep.ID = "Step" + i;
                objStep.Title = "Step " + i;
                TextBox objText = new TextBox();
                objText.ID = "Text" + i;
                objText.Text = "Value for step " + i;
                objStep.Controls.Add(objText);
                objWizard.WizardSteps.Add(objStep);
            }

            Controls.Add(objWizard);
        }
    }
}

```

Now that we have created a web part, we are ready for the next step, the creation of a SharePoint solution file. In the first part of the process, we will add a manifest file (Manifest.xml) to our web part library solution. The manifest file is used by Visual Studio 2005 and defines how to populate the cabinet file that is used to deploy the MyWebPart web part. The next procedure explains how to create a Manifest.xml file and how to add IntelliSense support to it:

1. In the MyWebPartLibrary project, choose Add ► New Item. This opens the Add New Item - WPLibrary dialog box.
2. In the Templates section, choose XML file.
3. In the Name text box, add the following value: **Manifest.xml**.
4. Click the Add button.
5. Click the Code window of Manifest.xml.
6. In the Properties window, click Schemas.
7. Click the ... button. This opens the XSD Schemas dialog box.
8. Click the Add button. This opens the Open XSD Schema dialog box.
9. Locate wss.xsd ([drive letter]:\Program Files\Common Files\Microsoft Shared\web server extensions\12\TEMPLATE\XML) and choose Open.

10. Click OK.
11. Add the code from Listing 1-11 to Manifest.xml:

**Listing 1-11.** *Creating a Manifest File*

```
<?xml version="1.0" encoding="utf-8" ?>
<Solution xmlns="http://schemas.microsoft.com/sharepoint/"
SolutionId="312ae869-37dc-4a74-9ecc-359fb3c1461d">
<Assemblies>
  <Assembly DeploymentTarget="WebApplication"
    Location="WPLibrary.dll">
    <SafeControls>
      <SafeControl Assembly="WPLibrary,
        Version=1.0.0.0,
        Culture=neutral,
        PublicKeyToken=9f4da00116c38ec5"
        Namespace="LoisAndClark.MyWebPartLibrary"
        TypeName="*"
        Safe="True" />
    </SafeControls>
  </Assembly>
</Assemblies>
</Solution>
```

After adding the Manifest.xml file to the web part library project, you can go ahead and create a cabinet project to deploy the MyWebPart web part to another SharePoint server or SharePoint web application. The next procedure explains how to create a SharePoint solution file:

1. Open the WPLibrary web part library solution.
2. Choose File ► Add ► New Project. This opens the Add New Project dialog box.
3. In the Project types section, expand the Other Project Types node and choose Setup and Deployment.
4. In the Templates section, choose CAB Project.
5. In the Name text box, add the following value: **WPLibraryCab**.
6. Click OK.
7. Right-click WPLibraryCab and choose Add ► Project Output. This opens the Add Project Output Group dialog box.
8. In the Project drop-down list, choose WPLibrary.
9. Select Primary output and Content Files.
10. Click OK.
11. Build WPLibraryCab.
12. Locate WPLibraryCab.cab and rename it to WPLibraryCab.wsp.
13. Open a command prompt, and type the following command:

```
stsadm -o addsolution -filename wplibrary.wsp
```

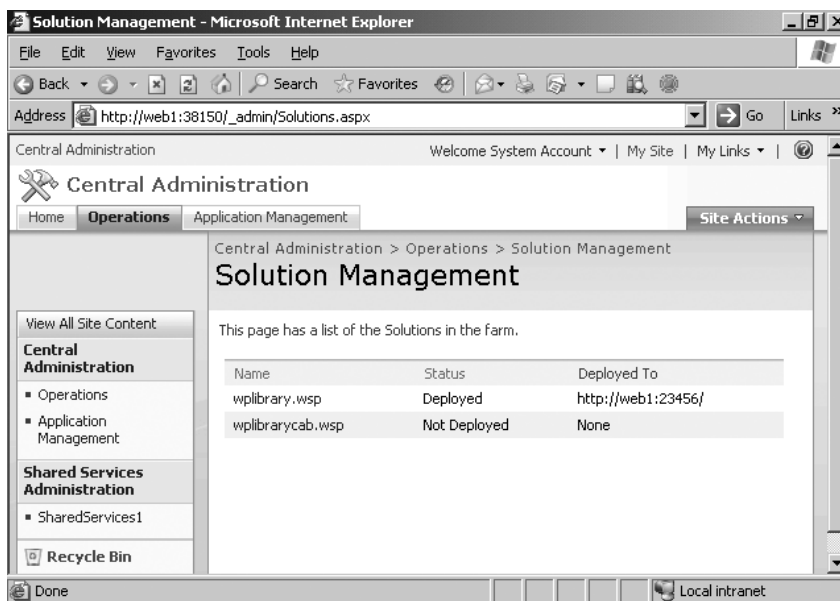
---

**Note** Make sure the account you are using to run this account has SharePoint farm administration rights and at least read/write access to the SharePoint content database. Otherwise, you will get the following error message: “Object reference not set to an instance of an object.”

---

Now, the SharePoint solution is added to SharePoint. You can deploy it further using the stsadm tool, or deploy the solution via the user interface of the SharePoint 3.0 Central Administration tool. The next procedure explains how to deploy the solution using the SharePoint 3.0 Central Administration tool:

1. Open SharePoint 3.0 Central Administration.
2. Click Operations. This opens the Operations page.
3. In the Global Configuration section, choose Solution Management. This opens the Solution Management page. This is shown in Figure 1-18.



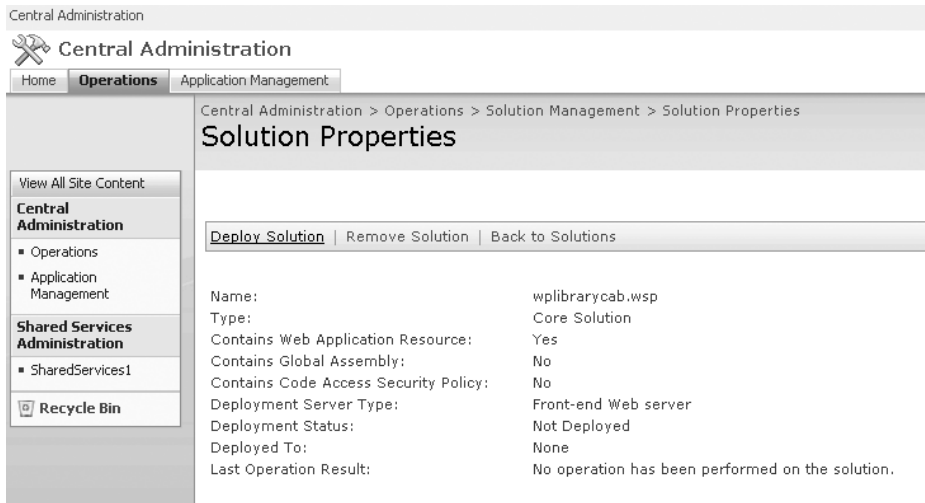
**Figure 1-18.** The Solution Management page shows a list of all available SharePoint solutions.

---

**Note** The Solution Management page contains an overview of all available solutions, and indicates whether they have been deployed or not. Figure 1-18 shows two solutions: wplibrary.wsp and wplibrarycab.wsp. Wplibrary.wsp was created and deployed automatically by Visual Studio 2005 extensions for Windows SharePoint Services 3.0 (discussed in the section “WSS 3.0 Tools: Visual Studio 2005 Extensions”). The other solution, wplibrarycab.wsp, is the one created in this section. As you can see, it is not deployed yet.

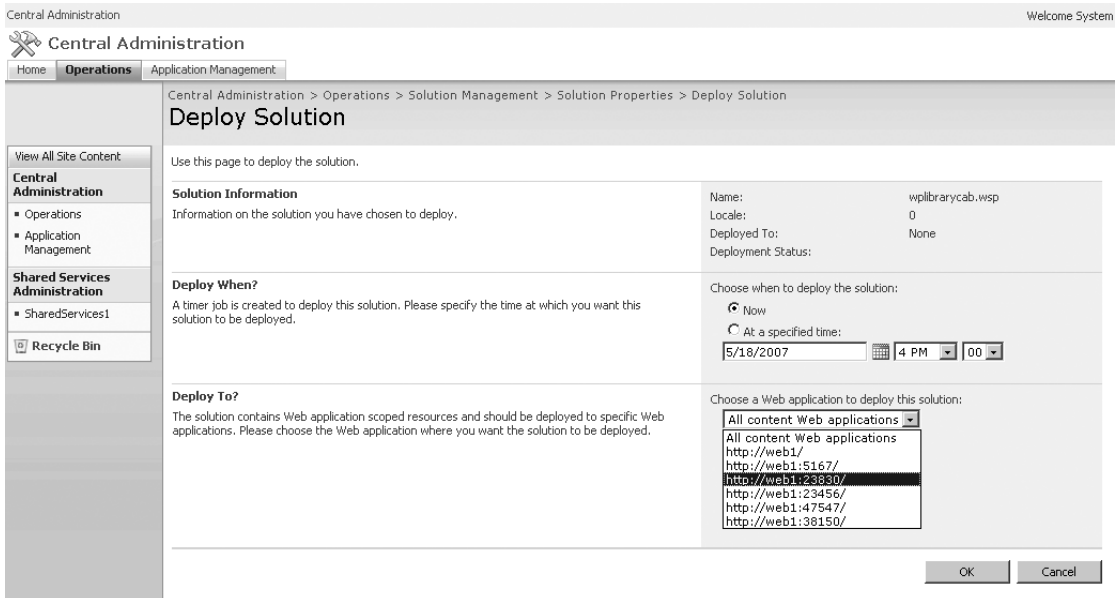
---

4. Click the wplibrarycab.wsp link. This opens the Solution Properties page, which is shown in Figure 1-19.



**Figure 1-19.** The *Solution Properties* page shows detailed information about a given SharePoint solution.

5. Click **Deploy Solution**. This opens the *Deploy Solution* page, shown in Figure 1-20.



**Figure 1-20.** The *Deploy Solution* page allows you to deploy a solution to a specific web application.

6. In the **Deploy to?** section, choose a Web application to deploy this solution.
7. Click **OK**. This opens the *Solution Management* page that is shown in Figure 1-21.



**Figure 1-21.** The *wplibrarycab.wsp* SharePoint solution has been deployed.

At this point, the SharePoint solution is deployed to a specific SharePoint server and a specific SharePoint web application. Before you are able to use the web parts deployed using a SharePoint solution, you need to add them to the web part gallery of the site collection:

8. Browse to the web application you have chosen in the previous step.
9. Choose Site Actions ► Site Settings. This opens the Site Settings page.
10. In the Galleries section, click Web Parts. This opens the Web Part Gallery page.
11. Click New. The web parts in the MyWebPartLibrary assembly should be listed; select them and click the Populate Gallery button.

You have seen how to create a SharePoint solution (.wsp) file that can be used to deploy web parts. After completing all the steps described in this section, you are ready to start using the web parts deployed via the SharePoint solution.

## Enhancing Development of Web Parts with the Guidance Automation Toolkit

Since the release of Visual Studio 2005 extensions for Windows SharePoint Services 3.0 (see the earlier section “WSS 3.0 Tools: Visual Studio 2005 Extensions”), creating, deploying, and debugging ASP.NET 2.0 web parts in Visual Studio 2005 has become easy as can be. Although we imagine you will use Visual Studio 2005 extensions most of the time when doing web part development because of its ease of use, it has a major drawback: you cannot extend the functionality offered by Visual Studio 2005 extensions yourself. This means that you will have to live with the limitations of Visual Studio 2005 extensions. The following is an overview of the most common limitations that exist during web part development:

- Development using Visual Studio 2005 extensions is slow. Every time a solution is executed (by pressing F5) that contains a web part library built using Visual Studio 2005 extensions, it creates a SharePoint solution, and retracts and deletes the current version of the solution if it has been deployed before. After that, the new solution is deployed. Every web part is installed as a separate feature; during deployment, every feature that has been previously deployed is deactivated and uninstalled before it can be activated again. In the final step of a deployment, the web server is restarted using `iisreset`. Overall, this makes running or debugging a web part library project a time-consuming experience.
- Visual Studio 2005 extensions do not provide support for building SharePoint legacy web parts (see the section “Web Parts Overview” for detailed information about SharePoint legacy web parts).

- Visual Studio 2005 extensions obscure the creation of a .webpart file that defines metadata about a web part. The SharePoint tab of the Project properties window offers some control over the configuration of your web part library, but it does not allow you to configure advanced .webpart elements.
- A web part built using the web part template in Visual Studio 2005 extensions overrides the `RenderContents()` method, which removes client-side IDs from your controls. As a result, you cannot use client-side code in a web part.
- The web part library assembly is always deployed to the Global Assembly Cache (GAC). Instead, our preference would be to place this assembly in the bin folder of a SharePoint web application. Deploying web part libraries to the Global Assembly Cache makes it impossible to define custom security policies and limit the code access permissions granted to those assemblies.
- Every web part in a web part library is deployed in a separate feature. Instead, you might be interested in deploying a group of web parts as a single unit.

Visual Studio 2005 extensions makes it easy to create web parts, at the cost of having granular control over a web part library project. Whether you want to put in the time and effort to create a custom solution that is more advanced than what is offered by Visual Studio 2005 extensions is completely up to you, although we doubt that we will ever go that road ourselves.

If you want to remain in control and are interested in facilitating web part development, you can create your own web part library template. The use of the Guidance Automation Toolkit (GAT) is paramount in achieving this goal.

The GAT is an extension to Visual Studio 2005 that allows architects to automate the easy parts of development so that developers can concentrate on the other parts. The GAT can be used to create assets that are developed in-house or by third parties, such as Microsoft. In the case of building a web part, the GAT can be used to build a package you can use as a template to start making SharePoint legacy web parts in Visual Studio 2005. It can also be used to adjust Visual Studio 2005 to your specific needs for every project type. In this section, you will learn how to use the Guidance Automation Toolkit to create a template that makes creating web part libraries easier.

## Guidance Automation Toolkit

In this section, we will show you how to create a web part library template using the GAT. If you want to work with the GAT, you first have to install the Guidance Automation Extensions, which can be downloaded at the Microsoft MSDN web site: <http://msdn.microsoft.com/vstudio/teamsystem/workshop/gat/>. Next, you should install the Guidance Automation Toolkit itself. You can find additional information about GAT at the following location: <http://msdn2.microsoft.com/en-us/teamsystem/aa718950.aspx>. In this book, we used the February 2007 CTP (Community Technology Preview) version of the GAT.

The GAT contains a couple of elements that work together to provide the automation functionality. The elements are recipes, actions, text template transformation templates, wizards, and Visual Studio templates, as described in the following list:

- *Recipes*: Recipes are used to automate activities that are usually performed by a developer. Most of the time, recipes automate following a series of instructions. Recipes are most suitable to automate repetitive actions.
- *Actions*: Actions are atomic units of work that are called in a defined order by recipes. The order in which actions are called is specified in the recipe definition. An action will accept input from an argument gathered by the recipe or from output received from another action that has already been run by the recipe.

- *Text template transformation templates:* Text template transformation templates (T4) are templates that contain a combination of text and *scriptlets*. Scriptlets are expressions in Visual Basic or C# that, when run, return a string that is directly inserted into the output stream of the template. The text template transformation engine is part of the GAT, and is also included in the Domain Specific Language (DSL) Toolkit, which we will discuss in Chapter 5. The templates are unfolded by this engine and the text part of a template is inserted unmodified in the template output. It is also possible to use an action to generate text from a T4 text template.
- *Wizards:* Wizards consist of a set of pages that help recipes to gather values. A recipe can have a wizard associated with it. A wizard can contain one or more steps. Steps are displayed as pages to guide the developer. Each page of the wizard will contain one or more fields. Each field is associated with a recipe argument, and includes a type converter and a user interface (UI) type editor. The type converter validates the value of a field and converts the value from a user interface representation to a type representation. The UI type editor is used to render the user interface that collects the information.

When the wizard displays a page, it will present the developer with a number of default values. These are initial values of arguments that are either set when the recipe reference is created or obtained by the recipe framework. When a recipe has value providers for all arguments—in other words, when a wizard already has default values—the user will still have to step through the wizard, which will show these values to the developer. This gives the developer a chance to acknowledge the default values.

It is possible that the user is not responsible for the input of a field value. Some fields may be filled in by the framework or the wizard itself. This process is known as *value propagation*. Fields that retrieve their values via value propagation will not be shown in the wizard. Value propagation is done via a combination of argument value change notifications and value providers observing the changes of other arguments (event handlers). As the user edits a field, the field value is validated and assigned to one or more corresponding arguments (parameters) that have registered their interest via value providers. After assignment, arguments are passed to a template.

In some cases, some fields of a wizard are required. The wizard framework provided with the GAT requires the user to populate fields that correspond to required arguments. The wizard will not enable the Next or Finish buttons on a page, nor will it allow you to access later pages in the wizard from the wizard sidebar, until all fields corresponding to required arguments contain data. If an argument is not collected at all by a wizard, and the associated value provider cannot find a value, the recipe framework will not initiate actions and instead will raise an exception.

- *Visual Studio templates:* Visual Studio templates are written in XML and are used by Visual Studio to create solutions or add one or more projects or items to an existing solution. The templates are unfolded by the Visual Studio template engine. Using the GAT, you can associate Visual Studio templates with recipes. This association means that when a template is unfolded, the wizard extension calls the recipe to let it collect parameter values, also known as *arguments*. The arguments are used to execute actions that may further transform solution items created by the template. Guidance packages can be managed via the Guidance Package Manager in Visual Studio 2005. You can find the Guidance Package Manager via the Tools menu. Once a guidance package is installed and enabled for a particular solution, recipes can be executed to carry out the required tasks.

## Creating the Web Part Library Template

In this section, you will learn how to create a web part library guidance package that makes creating SharePoint legacy web parts considerably easier in Visual Studio 2005. If you have not done so already, install the Guidance Automation Extensions and the Guidance Automation Toolkit.

---

**Caution** Installing the Guidance Automation Toolkit can take a considerable amount of time. Do not abort the installation process.

---

The GAT contains a predefined solution for developing a guidance package, which we will use to create a web part library template. The next procedure discusses how to create a guidance package:

1. Start Visual Studio 2005.
2. Choose File ► New ► Project. This opens the New Project dialog window.
3. Expand the Guidance Packages node.
4. Select the Guidance Package Development node.
5. In the templates section, select the Guidance Package template.
6. In the Name text field, enter the following value: **WebPartLibrary**.
7. In the Location text field, enter the following value: C:\projects.
8. Click OK. This opens the Create New Guidance Package window.
9. In the Package Description text field, enter the following text: **Package for creating a web part library**.
10. In the Package Namespace text field, enter the following value: **LoisAndClark.WebPartLibrary**.
11. Click Finish.

The WebPartLibrary guidance package is created. A guidance package contains lots of files and folders; throughout this section, you will learn how to develop a guidance package solution that facilitates the creation of SharePoint legacy web parts.

The first step for building a web part library template is to create a new folder in the Templates ► Solutions ► Projects folder. We will call this folder WebPartLibrary. You need to add the following (empty) files to this folder: WebPart.cs, WebPartLibrary.csproj, and a WebPartLibrary.vstemplate file. Then, you need to create a subfolder called Properties and add an empty file to the Properties folder called assembly.info. Now, you are finished with the initial setup of the WebPartLibrary guidance package.

### Adding a Solution Template

The WebPartLibrary.vstemplate file is a Visual Studio template that looks identical to a normal Visual Studio template except that it contains additional information used by the recipe framework. The template includes a <WizardExtension> element that specifies a class in the recipe framework that implements template extensions for the GAT. The XML in Listing 1-12 shows the content you need to add to the WebPartLibrary Visual Studio template (WebPartLibrary.vstemplate) in the WebPartLibrary folder:



**Listing 1-12.** *WebPartLibrary.vstemplate*

```
<VSTemplate Version="2.0.0" Type="Project" ➡
xmlns="http://schemas.microsoft.com/developer/vstemplate/2005">
  <TemplateData>
    <Name>WebPart Library Project</Name>
    <Description>WebPart class library project </Description>
    <Icon Package="{FAE04EC1-301F-11d3-BF4B-00C04F79EFBC}" ID="4547" />
    <ProjectType>CSharp</ProjectType>
    <SortOrder>20</SortOrder>
    <CreateNewFolder>>false</CreateNewFolder>
    <DefaultName>ClassLibrary</DefaultName>
    <ProvideDefaultName>>true</ProvideDefaultName>
  </TemplateData>
  <TemplateContent>
    <Project File="WebPartLibrary.csproj" ReplaceParameters="true">
      <ProjectItem ReplaceParameters="true">Properties\AssemblyInfo.cs</ProjectItem>
      <ProjectItem ReplaceParameters="true">WebPart.cs</ProjectItem>
    </Project>
  </TemplateContent>
  <WizardExtension>
    <Assembly>
      Microsoft.Practices.RecipeFramework.VisualStudio, Version=1.0.60429.0, ➡
      Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
    </Assembly>
    <FullClassName>
      Microsoft.Practices.RecipeFramework.VisualStudio.Templates.UnfoldTemplate
    </FullClassName>
  </WizardExtension>
</VSTemplate>
```

This XML is called a *solution template*. Solution templates are launched via the New command on the File menu.

**Adding a Project Template**

The next file in the package is the WebPartLibrary.csproj file. This is a project template that contains a project description. The template creates (or unfolds) a new project in an existing solution. The project template will include references to other assemblies and will be responsible for creating any class files, the project file (.csproj), and the assembly.info file. The XML in Listing 1-13 needs to be added to the project template (WebPartLibrary.csproj) and contains code that can be used to create the web part library project:

**Listing 1-13.** *The Project Template*

```
<Project DefaultTargets="Build" ➡
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProductVersion>8.0.30703</ProductVersion>
    <SchemaVersion>2.0</SchemaVersion>
    <ProjectGuid>$guid1$</ProjectGuid>
    <OutputType>Library</OutputType>
    <AppDesignerFolder>Properties</AppDesignerFolder>
    <RootNamespace>$safeprojectname$</RootNamespace>
    <AssemblyName>$safeprojectname$</AssemblyName>
  </PropertyGroup>
```

```

<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
  <DebugSymbols>true</DebugSymbols>
  <DebugType>full</DebugType>
  <Optimize>>false</Optimize>
  <OutputPath>\inetpub\wwwroot\bin\</OutputPath>
  <DefineConstants>DEBUG;TRACE</DefineConstants>
  <ErrorReport>prompt</ErrorReport>
  <WarningLevel>4</WarningLevel>
</PropertyGroup>
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
  <DebugType>pdbonly</DebugType>
  <Optimize>>true</Optimize>
  <OutputPath>\inetpub\wwwroot\bin\</OutputPath>
  <DefineConstants>TRACE</DefineConstants>
  <ErrorReport>prompt</ErrorReport>
  <WarningLevel>4</WarningLevel>
</PropertyGroup>
<ItemGroup>
  <Reference Include="System"/>
  <Reference Include="System.Data"/>
  <Reference Include="System.Xml"/>
  <Reference Include="System.Web"/>
  <Reference Include="Microsoft.SharePoint"/>
</ItemGroup>
<ItemGroup>
  <Compile Include="WebPart.cs" />
  <Compile Include="Properties\AssemblyInfo.cs" />
</ItemGroup>
<Import Project="$(MSBuildBinPath)\Microsoft.CSHARP.Targets" />
</Project>

```

## Defining the Final Part of the Web Part Library Guidance Package

The project template allows you to create new web part classes based on the web part template specified in WebPart.cs. Add Listing 1-14 to the WebPart.cs file:

### Listing 1-14. The Web Part Item Template

```

using System;
using System.ComponentModel;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Xml.Serialization;
using Microsoft.SharePoint;
using Microsoft.SharePoint.Utilities;
using Microsoft.SharePoint.WebPartPages;

namespace $safeprojectname$
{
  [DefaultProperty("Text"),
  ToolboxData("<{0}:$ClassName$ runat=server></{0}:$ClassName$>"),
  XmlRoot(Namespace = "$safeprojectname$")]
  public class $ClassName$ : WebPart
  {
    protected override void RenderWebPart(HtmlTextWriter output)

```

```

    {
        string htmlcode = "Hello World";
        output.Write(SPEncode.HtmlEncode(htmlcode));
    }
}

```

All arguments preceded by a \$ will be replaced once the project is created. Some values are entered in the wizard by the developer, and some values will have dynamic values based on other values. For example, we have decided that the default namespace of a solution will be identical to the solution name. The assembly.info file is another file that will be added to a project once a web part library project is created. It contains general information about the web part library assembly, and looks similar to WebPart.cs in that it uses arguments preceded by \$. Add Listing 1-15 to assembly.info:

**Listing 1-15.** *Assembly.info*

```

using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: AssemblyTitle("$projectname$")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("$registeredorganization$")]
[assembly: AssemblyProduct("$projectname$")]
[assembly: AssemblyCopyright("Copyright © $registeredorganization$ $year$")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: Guid("$guid1$")]

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]

```

Not only will we define a new project template, we also want to add a new item template that makes adding items that are related to the development of SharePoint legacy web parts easier. Create a new Visual Studio template in the Templates ► Items folder called WebPartTemplate.vstemplate (at this point, just an empty file). Then, add a new item template called WebPart.dwp. Leave the item template empty for now. This item template is responsible for creating a .dwp file. DWP files contain information about web parts. Add the XML in Listing 1-16 to the Visual Studio template (WebPartTemplate.vstemplate):

**Listing 1-16.** *Adding a Web Part (DWP) Item Template to the WebPartTemplate.vstemplate*

```

<VSTemplate Version="2.0.0" Type="Item" ➡
xmlns="http://schemas.microsoft.com/developer/vstemplate/2005">
  <TemplateData>
    <Name>Web Part Dwp</Name>
    <Description>Web Part Description File</Description>
    <Icon Package="{FAE04EC1-301F-11d3-BF4B-00C04F79EFBC}" ID="4515" />
    <ProjectType>CSharp</ProjectType>
    <SortOrder>10</SortOrder>
    <DefaultName>WebPart.dwp</DefaultName>
  </TemplateData>
  <TemplateContent>
    <ProjectItem ReplaceParameters="true">WebPart.dwp</ProjectItem>
  </TemplateContent>

```

```

<WizardExtension>
  <Assembly>
    Microsoft.Practices.RecipeFramework.VisualStudio, Version=1.0.60429.0, ➤
    Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
  </Assembly>
  <FullClassName>
    Microsoft.Practices.RecipeFramework.VisualStudio.Templates.UnfoldTemplate
  </FullClassName>
</WizardExtension>
<WizardData>
  <Template xmlns=http://schemas.microsoft.com/pag/gax-template ➤
    SchemaVersion="1.0" Recipe="NewItemClass"/>
</WizardData>
</VSTemplate>

```

The item template (WebPart.dwp) is used when the developer wants to add a .dwp file to the project. The item template uses a recipe called NewItemClass, which is discussed later. Add Listing 1-17 to the item template (WebPart.dwp):

**Listing 1-17. The Web Part Description Item Template**

```

<?xml version="1.0" encoding="utf-8"?>
<WebPart xmlns="http://schemas.microsoft.com/WebPart/v2">
  <Title>$TitleWebPart$</Title>
  <Description>$DescriptionWebPart$</Description>
  <Assembly>$AssemblyName$</Assembly>
  <TypeName>$Namespace$. $ClassName$</TypeName>
</WebPart>

```

All arguments preceded by a \$ are replaced once the developer creates a new .dwp file. Now you have seen the solution template, the project template, the item template, and all corresponding files. The last file that needs to be changed, which we have not discussed yet, is WebPartLibrary.xml.

You can find WebPartLibrary.xml directly underneath the project node in the Solution Explorer (the root folder of the WebPartLibrary project). This file contains the XML configuration code, containing all the recipes, actions, and wizards that are relevant within this guidance package. The first part of the WebPartLibrary.xml file contains a recipe called BindingRecipe. The <Action> element contains a reference to the item template called WebPartTemplate.vstemplate. The <Arguments> element contains all arguments that are asked for in the first wizard page when you create the solution. Listing 1-18 shows the contents of WebPartLibrary.xml:

**Listing 1-18. WebPartLibrary.xml**

```

<Recipe Name="BindingRecipe">
  <Types>
    <TypeAlias Name="RefCreator" ➤
      Type="Microsoft.Practices.RecipeFramework.Library.Actions. ➤
      CreateUnboundReferenceAction, Microsoft.Practices.RecipeFramework.Library"/>
  </Types>
  <Caption>Creates unbound references to the guidance package</Caption>
  <Actions>
    <Action Name="CreateSampleUnboundItemTemplateRef" Type="RefCreator" ➤
      AssetName="Items\WebPartTemplate.vstemplate" ➤
      ReferenceType="WebPartLibrary.References.ClassLibraryReference, ➤
      WebPartLibrary" />
  </Actions>
</Recipe>

```

```

<Recipe Name="CreateSolution">
  <Caption>Collects information for the new sample solution.</Caption>
  <Arguments>
    <Argument Name="ProjectName">
      <Converter Type="Microsoft.Practices.RecipeFramework.
        Library.Converters.NamespaceStringConverter,
        Microsoft.Practices.RecipeFramework.Library"/>
    </Argument>
    <Argument Name="ClassName">
      <Converter Type="Microsoft.Practices.RecipeFramework.
        Library.Converters.NamespaceStringConverter,
        Microsoft.Practices.RecipeFramework.Library"/>
    </Argument>
  </Arguments>
  <GatheringServiceData>
    <Wizard xmlns="http://schemas.microsoft.com/pag/gax-wizards"
      SchemaVersion="1.0">
      <Pages>
        <Page>
          <Title>Initial values for the new solution</Title>
          <Fields>
            <Field Label="Project Name" ValueName="ProjectName" />
            <Field Label="Class Name" ValueName="ClassName" />
          </Fields>
        </Page>
      </Pages>
    </Wizard>
  </GatheringServiceData>
</Recipe>

```

The next recipe (shown in Listing 1-19) that is important within the web part library template is called `NewItemClass`. This recipe contains all arguments that are collected by the wizard when you create a .dwp file. Some arguments have default values, such as assembly name and namespace.

#### Listing 1-19. *Recipe for the New Item Class*

```

<Recipe Name="NewItemClass" Recurrent="true">
  <xi:include href="TypeAlias.xml" xmlns:xi="http://www.w3.org/2001/XInclude" />
  <Caption>Collects information from the user</Caption>
  <Description></Description>
  <HostData>
    <Icon ID="1429"/>
    <CommandBar Name="Project" />
  </HostData>
  <Arguments>
    <Argument Name="CurrentProject" Type="EnvDTE.Project, EnvDTE,
      Version=8.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a">
      <ValueProvider Type="Microsoft.Practices.RecipeFramework.
        Library.ValueProviders.FirstSelectedProject,
        Microsoft.Practices.RecipeFramework.Library" />
    </Argument>
    <Argument Name="Namespace">
      <Converter Type="Microsoft.Practices.RecipeFramework.
        Library.Converters.NamespaceStringConverter,
        Microsoft.Practices.RecipeFramework.Library"/>
      <ValueProvider Type="Evaluator" Expression=
        "$ (CurrentProject.Properties.Item('DefaultNamespace').Value)" />
    </Argument>
  </Arguments>

```

```

    <Argument Name="AssemblyName">
      <Converter Type="Microsoft.Practices.RecipeFramework. ➤
        Library.Converters.NamespaceStringConverter, ➤
        Microsoft.Practices.RecipeFramework.Library" />
      <ValueProvider Type="Evaluator" Expression="$(CurrentProject.Name)" />
    </Argument>
    <Argument Name="ClassName">
      <Converter Type="Microsoft.Practices.RecipeFramework. ➤
        Library.Converters.NamespaceStringConverter, ➤
        Microsoft.Practices.RecipeFramework.Library"/>
    </Argument>
    <Argument Name="TitleWebPart">
      <Converter Type="Microsoft.Practices.RecipeFramework. ➤
        Library.Converters.NamespaceStringConverter, ➤
        Microsoft.Practices.RecipeFramework.Library"/>
    </Argument>
    <Argument Name="DescriptionWebPart">
      <Converter Type="Microsoft.Practices.RecipeFramework. ➤
        Library.Converters.NamespaceStringConverter, ➤
        Microsoft.Practices.RecipeFramework.Library"/>
    </Argument>
  </Arguments>
  <GatheringServiceData>
    <Wizard xmlns="http://schemas.microsoft.com/pag/gax-wizards" ➤
      SchemaVersion="1.0">
      <Pages>
        <Page>
          <Title>
            Collect information using editors, converters and value providers.
          </Title>
          <Fields>
            <Field ValueName="AssemblyName" Label="Assembly Name">
              <Tooltip></Tooltip>
            </Field>
            <Field ValueName="Namespace" Label="Namespace">
              <Tooltip></Tooltip>
            </Field>
            <Field ValueName="ClassName" Label="Classname Web Part">
              <Tooltip></Tooltip>
            </Field>
            <Field ValueName="TitleWebPart" Label="Title Web Part">
              <Tooltip></Tooltip>
            </Field>
            <Field ValueName="DescriptionWebPart" Label="Description Web Part">
              <Tooltip></Tooltip>
            </Field>
          </Fields>
        </Page>
      </Pages>
    </Wizard>
  </GatheringServiceData>
</Recipe>

```

After customizing the code and creating your own templates, build the project and register it. You can do this by right-clicking the WebPartLibrary project file and selecting Register Guidance

Package. This launches a recipe that is associated with the guidance package. The recipe registers the package you are developing on your computer. Registration is a form of installation that you can perform without leaving the Visual Studio development environment. It is also possible to unregister the package; this will reverse the registration.

After registering the guidance package, you can open a new instance of Visual Studio to test the functionality of the package.

## Installing and Using the Web Part Library Template

In this section, we will show you how to install and use the web part library template. Download the WebPartLibrarySetup.msi file from our web site (<http://www.lcbridge.nl/download>). You can install the web part library template by double-clicking the Windows Installer package. Close all instances of Visual Studio .NET 2005 before installing the package. The package will install all assemblies related to the web part library template in a folder dedicated to the guidance package. The Guidance Automation Extensions do not support assemblies in the GAC and will not load assemblies located there, even if the assembly is explicitly referenced in the guidance package. Double-click the .msi file to install the package. This opens a pop-up window with a welcome text, as shown in Figure 1-22.



**Figure 1-22.** First step of the setup of the web part library template

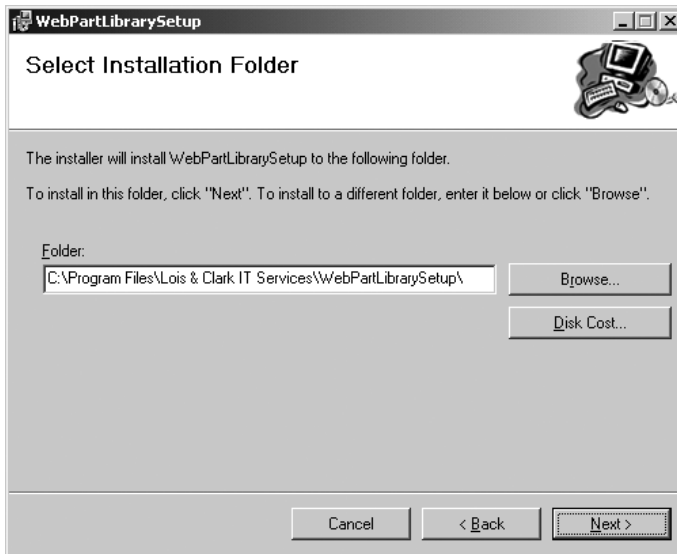
---

**Note** To create the Windows Installer package, we used the Guidance Automation Toolkit Technology February 2007 CTP version, because the official GAT was not released at the time of this writing.

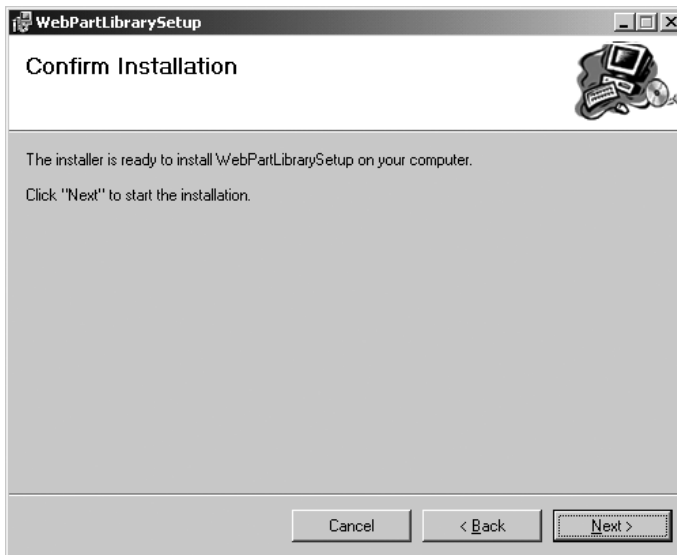
---

Next, choose the folder where you want the template to be installed and click Next. See Figure 1-23.

Click Next to start the installation, as shown in Figure 1-24.



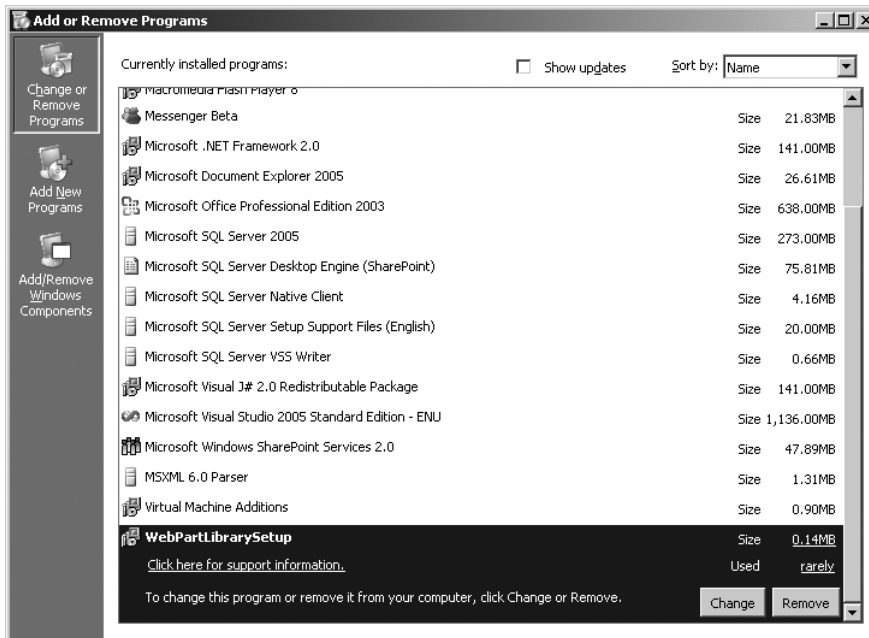
**Figure 1-23.** *Second step of the setup of the web part library template*



**Figure 1-24.** *Last step of the setup of the web part library template*

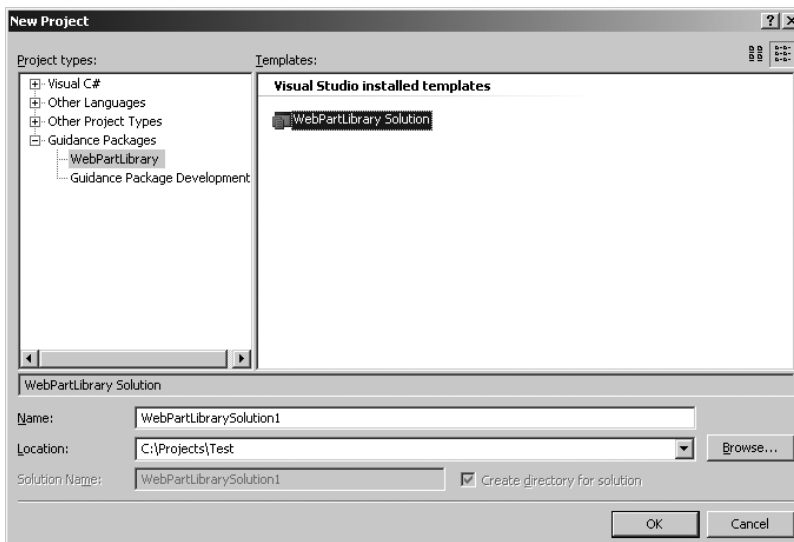
When the installation has succeeded, you will find the web part library template under Add and Remove Programs, as seen in Figure 1-25. This is where you can remove or repair the template. You cannot install two versions of the same guidance package at the same time. If you do attempt to install a guidance package with the same name as an existing guidance package, the Guidance Automation Extensions will throw an exception, informing you that you must uninstall the previous instance of the guidance package before installing the new one.





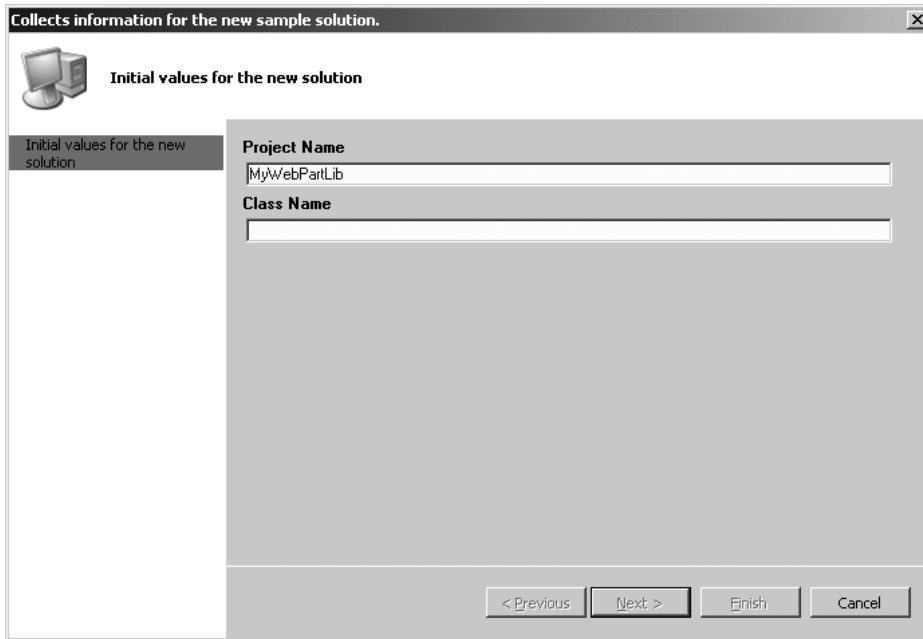
**Figure 1-25.** *The Add and Remove Programs window*

You will also find evidence of installing the web part library template in Visual Studio 2005. Open Visual Studio 2005 and choose **File ► New ► Project**. In the New Project window, you will see a new project type called **Guidance Packages**. Click **Guidance Packages**. Underneath it you will find a package called **WebPartLibrary**. In the right window pane, you will see a template called **WebPartLibrary Solution**, as shown in Figure 1-26.



**Figure 1-26.** *Choose a project in Visual Studio 2005.*

Click WebPartLibrary Solution, fill in a location and a descriptive name, and click OK. This will start a wizard page where you will fill in a project name and the name of the class that will be created initially. This is shown in Figure 1-27. After providing a project name and a class name, you can finish the wizard.

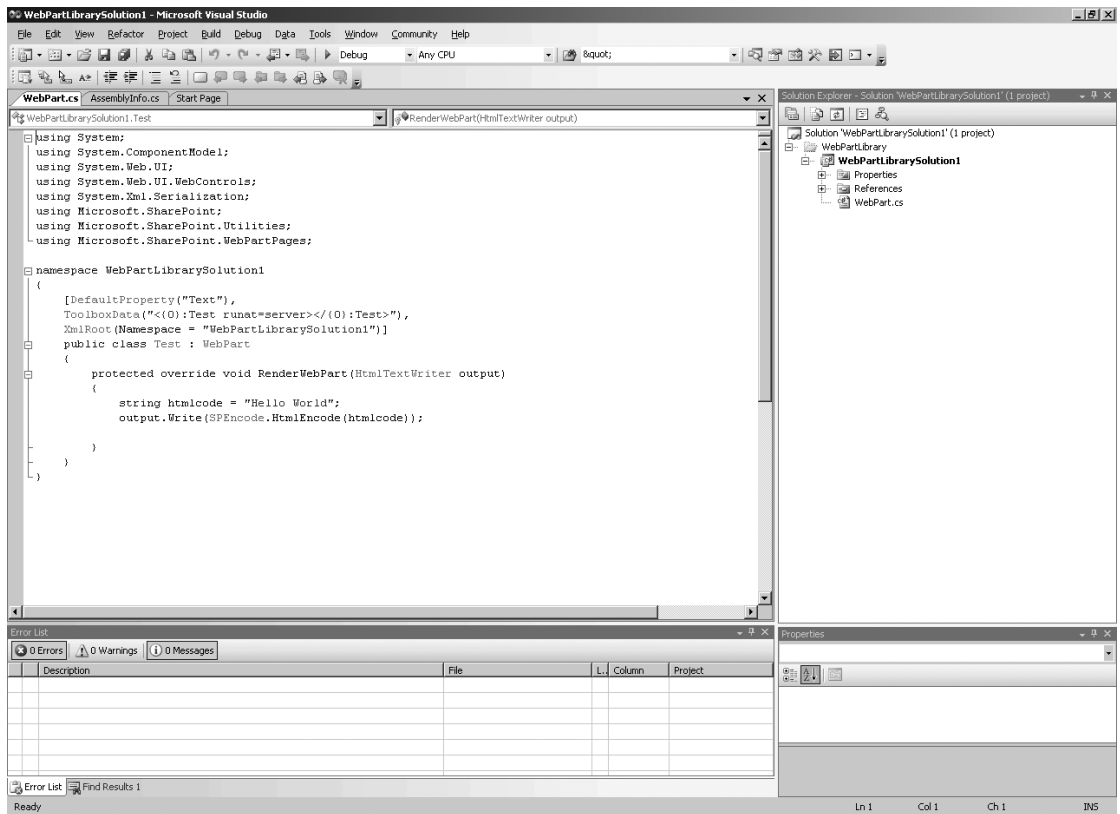


**Figure 1-27.** Wizard page belonging to the project creation

A couple of things will be created at this point:

- A solution containing a Properties directory with an assembly.info file
- A References directory containing the following references:
  - System
  - System.Data
  - System.Web
  - System.Xml
  - Microsoft.SharePoint
- A class file called WebPart.cs

Figure 1-28 shows the new solution created with the help of the web part library template.



**Figure 1-28.** Web part library template in Visual Studio

The `WebPart.cs` class contains the code shown in Listing 1-20:

**Listing 1-20.** Generated Code in a Web Part Created Using the Custom Web Part Library Template

```
using System;
using System.ComponentModel;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Xml.Serialization;
using Microsoft.SharePoint;
using Microsoft.SharePoint.Utilities;
using Microsoft.SharePoint.WebPartPages;

namespace WebPartLibrarySolution1
{
    [DefaultProperty("Text"),
    ToolboxData("<{0}:Test runat=server></{0}:Test>"),
    XmlRoot(Namespace = "WebPartLibrarySolution1")]
    public class Test : WebPart
    {

```

```
protected override void RenderWebPart(HtmlTextWriter output)
{
    string htmlcode = "Hello World";
    output.Write(SPEncode.HtmlEncode(htmlcode));
}
}
```

The class itself inherits from the `Microsoft.SharePoint.WebPartPages.WebPart` class, which makes it a web part that is suitable to be used within a SharePoint site.

There are two things left to do. First, check whether the output path of the project is changed to the bin directory of the virtual server's root folder. If it is not, change the output path. This makes testing the web part considerably easier. Otherwise, you would have to copy the web part DLL manually each time you compile. By default, the root folder is [drive letter]\inetpub\wwwroot. If the bin folder does not exist, you have to create one yourself. Second, you need to add your assembly to the SafeControls list in the web.config file, which is also located in the root of the SharePoint web application.

```
<SafeControl Assembly="MyWebPartLib" Namespace="MyWebPartLib" TypeName="*"
Safe="True" />
```

Here, we are using a partially qualified assembly name, which is great for creating code examples. We could have chosen to use fully qualified names, a practice that is recommended for production code. Those names include the following information: assembly name, version number, culture (which is always set to neutral for code assemblies), and developer identity (a public key token).

Assemblies with fully qualified names are also known as *strong-named* assemblies. Strong-named assemblies make it easier to enforce security policies for assemblies, because you can assign security permissions based on developer identity. Another advantage is that strong names make creating unique assembly names easier, thus reducing the chances for name conflicts. The content of a strong-named assembly cannot be tampered with after compilation, as strong-named assemblies contain a hash code representing the binary content of the assembly. This hash code is unique for every assembly, and the .NET CLR makes sure the hash code matches the assembly content during load time. If someone has tampered with your assembly after compilation, the hash code will not match the content and will not be loaded. A final advantage is that version policies are only applied to strong-named assemblies, not to assemblies with partially qualified names. Strong names are mandatory for assemblies that need to be installed in the GAC. The next code fragment shows a SafeControl entry for a strong-named assembly:

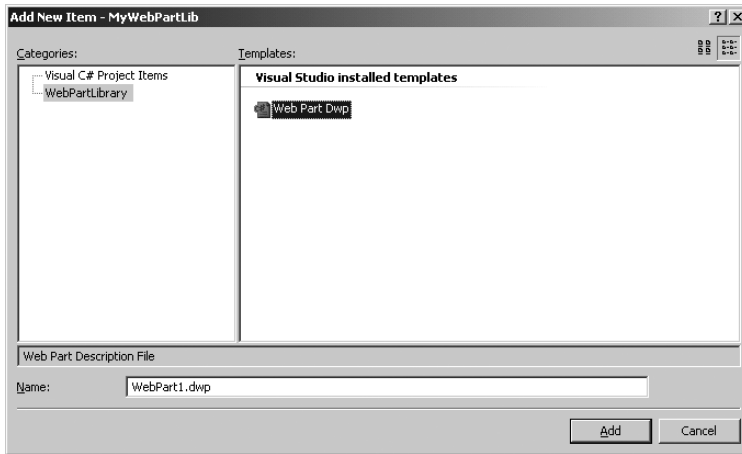
```
<SafeControl Assembly="MyWebPartLib" Namespace="MyWebPartLib,
Version=1.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429d" TypeName="*"
Safe="True" />
```

If you want to register your web part on a SharePoint site via a Web Part Description File, you can create one by right-clicking your project and choosing Add New Item. Under Categories, you will find a new category called WebPartLibrary. Click this category and choose the template called Web Part Dwp. Give the .dwp file a descriptive name and click Add. Figure 1-29 shows the Add New Item window.

---

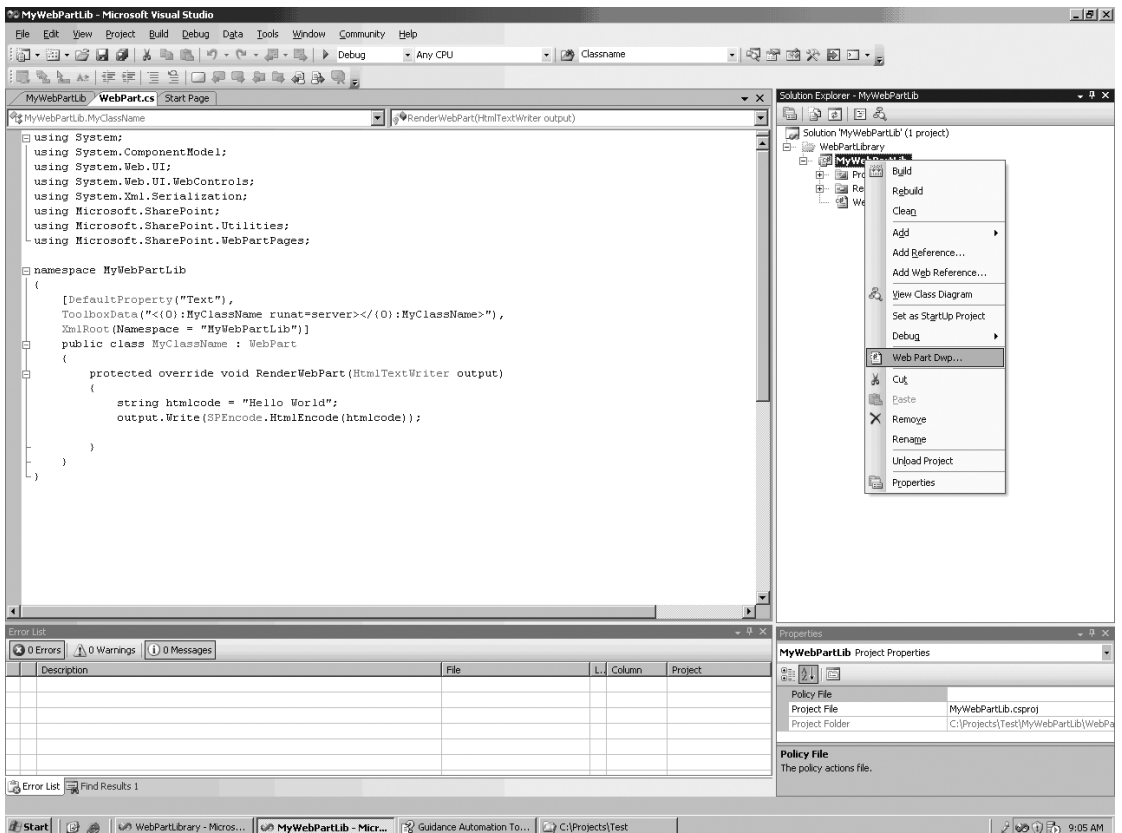
**Note** SharePoint 2003 used .dwp (Dashboard web part) files to describe web parts. In ASP.NET 2.0, similar XML files with the extension .webpart are used for the same purpose. This version of the web part library template does not contain an item template that corresponds to .webpart files, although such a template is easy to add. Microsoft Office SharePoint Server 2007 supports the use of both web part description formats. If you are using a (legacy) .dwp file to import a web part, the web part needs to derive from the `Microsoft.SharePoint.WebPartPages.WebPart` class. Failing to do so results in an error message when you try to add the web part to a web part zone.

---



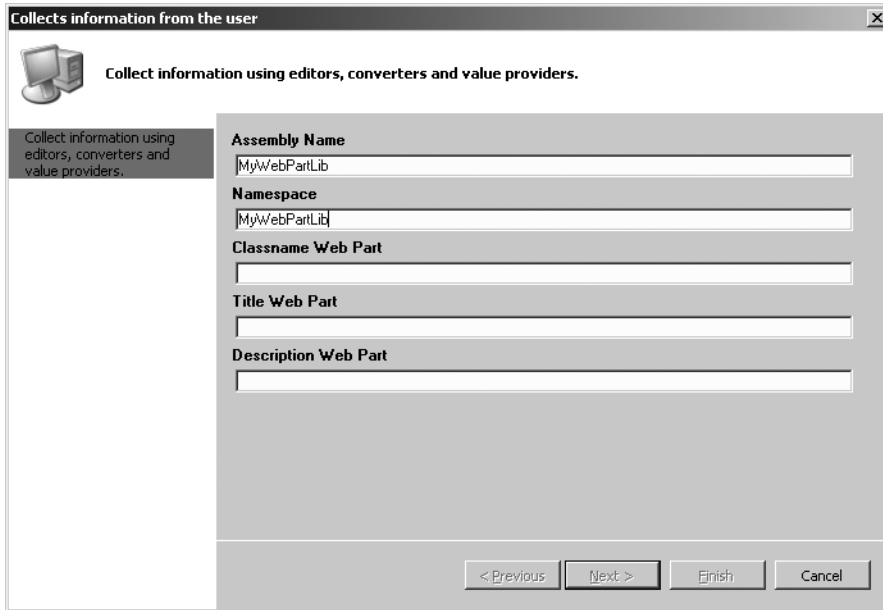
**Figure 1-29.** The Add New Item window in Visual Studio 2005

You can also create a .dwp file by right-clicking your project and then choosing Web Part Dwp, as you can see in Figure 1-30. This will open the Add New Item window as well.



**Figure 1-30.** Add a new web part .dwp file by right-clicking your project.

This will start a wizard that collects information about the Web Part Description file. The assembly name and namespace will have default values. You will only have to type the name of the class, a title for your web part, and a description. The title and description are shown on the SharePoint site page the web part is imported into. The wizard is shown in Figure 1-31.



Collects information from the user

Collect information using editors, converters and value providers.

Collect information using editors, converters and value providers.

**Assembly Name**  
MyWebPartLib

**Namespace**  
MyWebPartLib

**Classname Web Part**

**Title Web Part**

**Description Web Part**

< Previous   Next >   Finish   Cancel

**Figure 1-31.** *The web part .dwp file wizard*

At this point, you have created your first web part using the web part library template. Now you can start adding your own code to the web part.

## Summary

This chapter explored the incorporation of the new features of ASP.NET 2.0 into Microsoft Office SharePoint Server 2007. We talked about the integration between SharePoint and .NET Framework 2.0, and we provided an overview of the basic anatomy of web parts. Then, you learned how to create web parts in Visual Studio .NET 2005 using the new Visual Studio 2005 extensions for Windows SharePoint Services 3.0. Then, we showed you how to use a couple of the new ASP.NET 2.0 server controls within a web part. Finally, we discussed how the Guidance Automation Toolkit can be used to build a web part library template guidance package to enhance the creation of web parts.