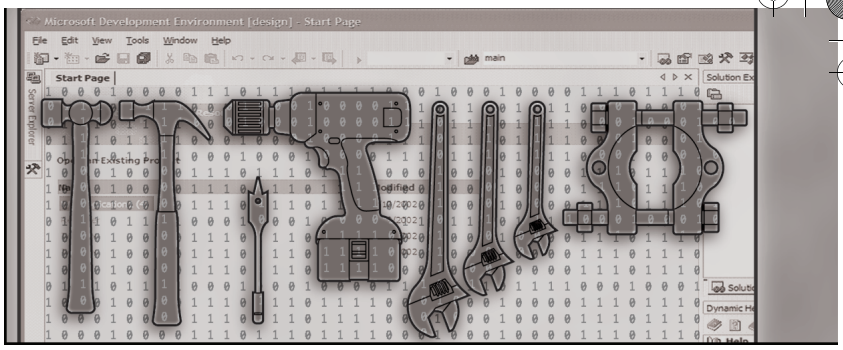

9

Visual Studio .NET Wizards

Wizards, which are familiar to users of Microsoft Windows, provide a simple, step-by-step way of making a complex task simple. In this chapter, we'll discuss how to create your own wizards to run within Microsoft Visual Studio .NET.

An Overview of Wizards

A programmer's work can be repetitive. There's plenty of new, innovative code to write, but a lot of code is common to all projects. Rather than writing this code over and over, you can use a wizard to generate the starter code and start writing the core implementation of a project. A wizard can display a dialog box to walk the user through a set of steps, asking questions, and it uses the answers to make a complicated or often-repeated task easier to complete. Alternatively, a wizard can skip displaying a dialog box and simply generate code without asking the user for any input. Windows is full of wizards, such as wizards that help connect to printers and networks and even ones to find help when something goes wrong. Visual Studio .NET, on the other hand, uses wizards to generate code.

Types of Wizards

You can build and run three types of wizards in Visual Studio .NET. The type that's probably the most familiar to developers is the New Project wizard. A New Project wizard, as its name suggests, generates the code for a project that gives the user a starting point for a new program. New Project wizards are invoked when the user selects an item in the right panel of the New Project dialog box, which is displayed by choosing File | New | Project.

The second type of wizard is an Add New Item wizard. Once a project has been created, a user often needs to add new files, such as classes, images, or Web pages, to that project. An Add New Item wizard can be used to create these new files. The common way to access this type of wizard is by right-clicking on a project in the Solution Explorer window and choosing Add | Add New Item. This displays the Add New Item dialog box, from which wizards can be run.

The third and least-often used wizard type is a Custom wizard. A Custom wizard isn't invoked directly by Visual Studio .NET; rather, it is explicitly called by a macro, an add-in, or another wizard. A Custom wizard can't be classified as an Add New Item wizard or a New Project wizard, but it can walk the user through a set of steps to accomplish some task. With a Custom wizard, you can add wizard-like functionality anywhere within Visual Studio .NET and not be limited only to creating new projects or adding new files to an existing project, as you are with New Project or Add New Item wizards.

Whether you choose to create a New Project, Add New Item, or Custom wizard, you implement it in the same basic way: you create a COM object that implements the wizard, you create a .vsz file to let Visual Studio .NET know about your wizard, and then you create the source code templates. We'll discuss each of these wizard types in this chapter as well as how to build them.

Creating the Wizard Object

Every wizard, whether it is a New Project wizard, an Add New Item wizard, or a Custom wizard, is simply a COM object that implements the *EnvDTE.IDTWizard* interface. *Execute*, the only method of this interface, is called when Visual Studio .NET loads the wizard. The signature for this interface is

```
public interface IDTWizard
{
    public void Execute(object Application, int hwndOwner,
        ref object[] ContextParams,
        ref object[] CustomParams,
        ref EnvDTE.wizardResult retval)
}
```

A number of arguments are supplied to the *Execute* method:

- ***Application*** The *DTE* object for the instance of Visual Studio .NET in which the wizard is being run
- ***hwndOwner*** A handle to a window that the wizard can use as a parent for any user interface elements that the wizard creates

- ***ContextParams*** An array of type *object* that describes the state Visual Studio .NET is in when the wizard is run
- ***CustomParams*** An array of type *object* containing data defined by the wizard writer and passed to the wizard object

The *Execute* method is where all the processing for a wizard takes place. Within this method, a wizard has complete control over how it performs its work. Visual Studio .NET places no restrictions on how a wizard is implemented other than that it must implement the *IDTWizard* interface on a COM object. A wizard can display a user interface to ask the user questions, or it can use the information provided through the various arguments of the *Execute* method to perform its work. You can think of the *Execute* method as similar to the *Main* function of a Visual Basic or Visual C# console application: once called, it can do whatever it wants.

The *ContextParams* argument passed to *Execute* is an array of elements that's populated with values the user enters in the Add New Item or New Project dialog box. It also contains a number of other values, provided by Visual Studio .NET, that give hints to your wizard about how it should generate code. The values in the array change depending on whether the wizard is run as a New Project wizard, an Add New Item wizard, or a Custom wizard. The arguments for the various project types and the order in which those types appear are listed in Table 9-1 and Table 9-2.

Table 9-1 *ContextParams* Array Values Passed to a New Project Wizard

Value	Description
Wizard Type	The value *EnvDTE.Constants.vsWizardNewProject*.
Project Name	The name of the project to create. This doesn't include the filename extension.
Local Directory	The directory in which to create the project or solution.
Installation Directory	The location on disk where Visual Studio .NET was installed.
Exclusive	If this value is *true*, a wizard should close the current solution and create a new one. If the value is *false*, the solution shouldn't be closed and the project should be added to the currently open solution file.

(continued)

Table 9-1 *ContextParams* Array Values Passed to a New Project Wizard *(continued)*

Value	Description
Solution Name	The name of the solution to create, if specified. This solution name is available if the Create Directory For Solution check box is selected in the New Project dialog box.
Silent	A Boolean flag indicating whether the wizard should run without displaying any user interface elements to the user. If this is *true*, use reasonable defaults when you generate code.

Table 9-2 *ContextParams* Array Values Passed to an Add New Item Wizard

Value	Description
Wizard Type	The value *EnvDTE.Constants.vsWizardAddItem*.
Project Name	The name of the project the item is being added to.
Project Items	The *EnvDTE.ProjectItems* collection the item should be added to.
New Item Location	The folder on disk in which the item should be created.
New Item Name	The name the user entered into the Name box in the Add New Item dialog box.
Product Install Directory	The folder in which the programming language is installed.
Silent	A Boolean flag indicating whether the wizard should run without displaying any user interface elements to the user. If this is *true*, use reasonable defaults when you generate code.

We didn't include a table that lists context parameters for a Custom wizard because these are not determined by Visual Studio .NET—they're supplied by an add-in, a macro, or even another wizard. We'll discuss the Custom wizard type and the *CustomParams* that Custom wizards are passed in more detail later in this chapter.

When run, a wizard should verify that the first element of the context parameter array is the constant *EnvDTE.Constants.vsWizardNewProject* if the wizard is a New Project wizard or the constant *EnvDTE.Constants.vsWizardAddItem* if the wizard is an Add New Item wizard. If the GUID doesn't match the type of wizard the object implements, the object should return the error code *wizardResult.wizardResultFailure* through the *retval* argument of *IDTWizard.Execute*.

> **Note** When you check the GUID that is passed as the wizard type, you should perform a case-insensitive comparison because the constants *vsWizardNewProject* and *vsWizardAddItem* might have a different case than any value passed from Visual Studio .NET as the first value in the *ContextParams* array.

An example implementation of a wizard and the code to extract the elements from the *ContextParams* array are shown in Listing 9-1.

Wizard.cs

```
using System;
using System.Runtime.InteropServices;

namespace BasicWizard
{
    [GuidAttribute("E5D0A8B2-A449-4d3b-B47B-99494D23A58B"),
    ProgIdAttribute("MyWizard.Wizard")]
    public class Wizard : EnvDTE.IDTWizard
    {
        public void Execute(object Application, int hwndOwner,
                            ref object[] ContextParams,
                            ref object[] CustomParams,
                            ref EnvDTE.wizardResult retval)
        {
            EnvDTE.DTE application = (EnvDTE.DTE)Application;
            string wizardType = (string)ContextParams[0];

            if(System.String.Compare(wizardType,
                EnvDTE.Constants.vsWizardNewProject, true) == 0)
            {
                string newProjectName = (string)ContextParams[1];
                string newProjectLocation = (string)ContextParams[2];
                string visualStudioInstallDirectory =
                     (string)ContextParams[3];
                bool exclusiveProject = (bool)ContextParams[4];
                string newSolutionName = (string)ContextParams[5];
                bool runSilent = (bool)ContextParams[6];
            }
            else if(System.String.Compare(wizardType,
                EnvDTE.Constants.vsWizardAddItem, true) == 0)
            {
```

Listing 9-1 The wizard add-in source code

```
                string projectName = (string)ContextParams[1];
                EnvDTE.ProjectItems projectItems =
                     (EnvDTE.ProjectItems)ContextParams[2];
                string newItemLocation = (string)ContextParams[3];
                string newItemName = (string)ContextParams[4];
                string productInstallDirectory = (string)ContextParams[5];
                bool runSilent = (bool)ContextParams[6];
            }
            else
            {
                //ERROR! Unknown wizard type
            }
        }
    }
}
```

Creating the .vsz File

As you saw in Chapter 6, to create an add-in you must provide information to Visual Studio .NET to let it know that the add-in is available to be loaded. This information, which is stored in the system registry, includes the programmatic identifier (ProgID) as well as information detailing how the add-in should be loaded. Likewise, a wizard needs a way to announce itself as being available; but unlike with an add-in, you must rely on the file system to make a wizard available. You do this by creating a hierarchy of folders in a specific location on disk and placing files with the extension .vsz in within this folder hierarchy.

A .vsz file has a simple text-based file format. The file starts with the string "*VSWIZARD 7.0*", which tells Visual Studio .NET that the file declares a wizard and that the wizard should be run in Visual Studio .NET version 7 or later. The next line of text is a token that starts with "*Wizard=*" and is followed by the ProgID or the class identifier (ClassID) of the COM object implemented by the wizard. If we were to use the ProgID from the Wizard.cs code shown in Listing 7-1, the line in the .vsz file would appear as follows:

```
Wizard=MyWizard.Wizard
```

We could also use the ClassID format:

```
Wizard={E5D0A8B2-A449-4d3b-B47B-99494D23A58B}
```

After the line for the ProgID or ClassID, you can place a list of user-defined data. This data can be any string data that you want to pass to your wizard, and

it can be static (hard-coded into the .vsz file during development) or generated when your wizard is installed by a setup program. Each line of this data starts with the token "*Param=*", and your wizard can require any number of these entries (including 0). Here's an example of this data:

```
Param=Hello World
Param=Second line of data
```

Each of these *Param* tokens is passed as an element of the *CustomParams* array when your wizard's *Execute* method is invoked and can be found within a wizard with code such as this C# snippet:

```
for(int i = 0 ; i < CustomParams.Length ; i++)
{
    string data = (string)CustomParams[i];
    System.Windows.Forms.MessageBox.Show(data);
}
```

When this code runs, the strings passed to *CustomParams* have the leading "*Param=*" stripped from each string; only the raw data is specified.

> **Note** Even if you're using Visual Studio .NET 2003 (version 7.1), the first line of a .vsz file must start with the string *'VSWIZARD 7.0'*, not *'VSWIZARD 7.1'*.

Where to Save .vsz Files

For a user to run your wizard, you must place the .vsz file in a specific location on disk so the New Project or Add New Item dialog box can find it and make that wizard available to be run. When the New Project or Add New Item dialog box is shown, a folder or number of folders on disk are read for the subfolders and files they contain. The names of these folders are inserted into the tree on the left side of the dialog box, and any subfolders are inserted as subitems of that tree node. As the user selects nodes in the tree, each file within the folder corresponding to the selected node is displayed in the list on the right side of the dialog box.

For example, Figure 9-1 shows the New Project dialog box with the file system modified to add a folder called A Sub Folder under the Extensibility Projects folder, which is the folder on disk where the .vsz files are stored for the Add-in Wizard. When the contents of the Extensibility Projects folder are copied into the A Sub Folder folder, they appear on the right side of the dialog box if

this new folder is selected. The A Sub Folder folder was created in the folder C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\Extensibility Projects (using the default installation location).

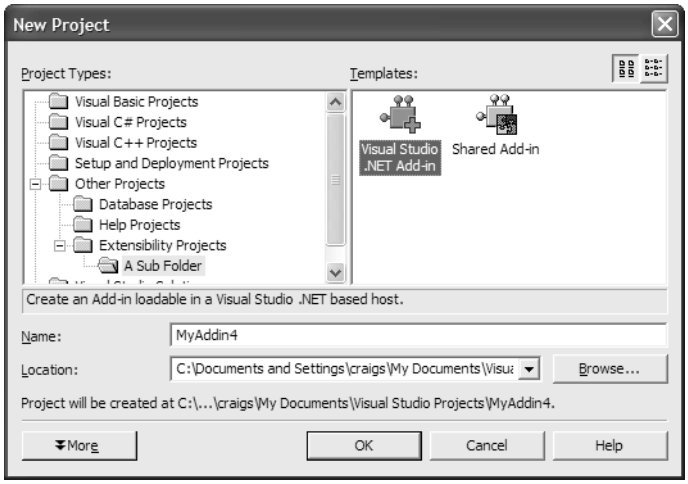

Figure 9-1 A new subfolder shown in the New Project dialog box

You can find the location to store your new project .vsz files programmatically using the *TemplatePath* property of the *Solution* object. The following macro displays message boxes showing the folder in which the .vsz files can be stored so that they will appear within the Visual Basic Projects and Visual C# Projects nodes on the right side of the New Project dialog box:

```
Sub VSZLocation()
    'Display the .vsz path for Visual Basic Projects
    MsgBox(DTE.Solution.TemplatePath( _
        VSLangProj.PrjKind.prjKindVBProject))

    'Display the .vsz path for C# Projects
    MsgBox(DTE.Solution.TemplatePath( _
        VSLangProj.PrjKind.prjKindCSharpProject))
End Sub
```

> **Note** The *TemplatePath* property was poorly named—a better name would be *VSZFilePath*. Don't confuse the word *Template* in the property name with file templates (which we'll discuss later in this chapter).

The constants *prjKindVBProject* and *prjKindCSharpProject*, which are defined in the metadata assembly VSLangProj.dll, are GUIDs in the form of a string. There are, as you probably know, project types other than those for Visual Basic and C#, but constants that can be passed to *TemplatePath* property for those projects types aren't found in any assembly. You can manually find the project type GUIDs for these other project types by poking around in the system registry. Under the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1\Projects is a list of GUIDs; each GUID defines a project type that Visual Studio .NET supports. Replacing the argument to *Solution.Template* path with one of these GUIDs returns the path in which to store your .vsz file so that an entry for the wizard appears in the New Project dialog box for that project type. If we search through this area of the registry for *vcproj* (the extension used for Visual C++ project files), we'll find that the GUID for the Visual C++ project type is {8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}. We can use this GUID to locate the path to where we can store .vsz files so they'll appear in the Visual C++ Projects node of the New Project dialog box:

```
Sub VSZLocation2()
    'Display the .vsz path for Visual C++ Projects:
    MsgBox(DTE.Solution.TemplatePath( _
        "{07CD18B1-3BA1-11d2-890A-0060083196C6}"))
End Sub
```

As you can see in Figure 9-1, a different image is shown for each .vsz file found. You can associate an image with a .vsz file by placing an icon (.ico) file in the same folder—one with the same name as the .vsz file but with the .ico extension. For the Add-in Wizard, the .vsz file on disk is called Visual Studio .NET Add-in.vsz. When the user selects the folder, a file called Visual Studio .NET Add-in.ico is searched for and, if found, used as the display image. If an icon for a .vsz file isn't found, the default icon for files as defined by Windows is used.

The Add New Item dialog box uses the directory structure in a similar way to the New Project dialog box. You can add new folders, and any files with a .vsz extension that the user selects will be run as a wizard. The only difference between the Add New Item dialog box and the New Project dialog box is that the template directories are located in different places. Figure 9-2 shows the directory structure after it was modified for the Add New Item dialog box and a text file template was placed in that folder.

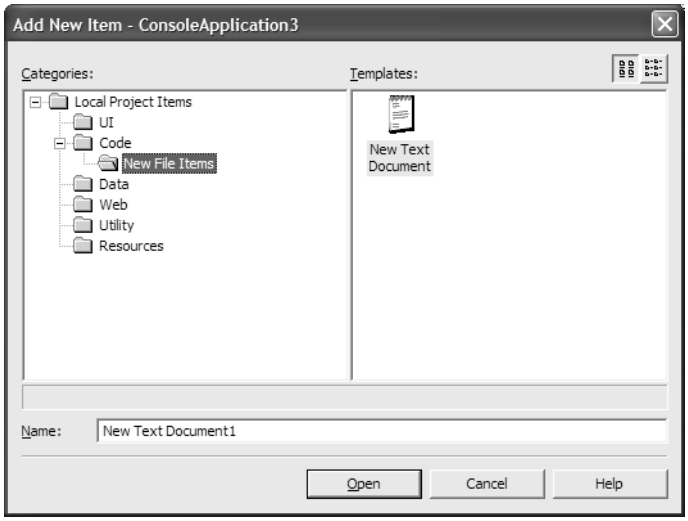

Figure 9-2 A custom folder in the Add New Item dialog box

You retrieve the location where the .vsz files are stored for this dialog box much like you retrieve the path for the New Project dialog box, but using a different method. Rather than using the *Solution.TemplatePath* method, you pass the project type GUID to the *Solution.ProjectItemsTemplatePath* method. You can use the following macro to find the path to where C# Add New Item .vsz files can be stored:

```
Sub ProjectItemVSZLocation()
    'Display the .vsz path for C# project items:
    MsgBox(DTE.Solution.ProjectItemsTemplatePath( _
        VSLangProj.PrjKind.prjKindCSharpProject))
End Sub
```

Creating Wizard Templates

A wizard's purpose is to create a new project or add code files to an existing project. But where does the code for these projects or project items come from? The answer is template files. Templates are the source code files that a wizard adds to a solution or an existing project. These files are placed on disk, and when a wizard wants to add the project or file, the template project and the files the project references or the file for an Add New Item wizard is copied into a folder the user specifies and is then added to the solution or project.

Templates are normally created using one of the wizards for generating a project or a project item, and then the file(s) of the new project are modified to

fit the requirements of the project or project item you're trying to create. The code that's created and added to a solution when you run the Add-in Wizard is generated in this way. We used the C# and Visual Basic Class Library Wizard to generate the base project and then modified this project to implement the add-in. The Add-in Wizard locates this project and adds it to the solution, and then the files in this newly created project are modified to conform to the options the user selected when running the wizard.

Using Template Files

Once you've created the template files, you need a way to add them to the solution or project. Visual Studio .NET supports a number of methods to accomplish this. In Chapter 8, we explored the project model but purposely left out a discussion of two methods of the *Solution* object: *AddFromFile* and *AddFromTemplate*. These two methods are used to add project templates to a solution. *AddFromFile* adds a reference in the solution file to the project, keeping the project file where it exists on disk. Calling this method is analogous to right-clicking on the solution node in the Solution Explorer window, choosing Add | Add Existing Project, and browsing to a project file. Wizards, however, usually want to add a copy of a project template to the solution; otherwise, the user of the generated project would modify the template project and subsequent running of the wizard would add a reference to this same modified project. Wizards should normally use the *AddFromTemplate* method, which copies the project template and its associated files to a destination folder and then adds a reference of this copy to the solution. The signature for *AddFromTemplate* is

```
public EnvDTE.Project AddFromTemplate(string FileName, string Destination,
    string ProjectName, bool Exclusive = false)
```

Here are the arguments:

- ***FileName*** The full path to the project template.
- ***Destination*** The location on disk to which the project and the files it references are copied. The wizard should create this destination path before *AddFromTemplate* is called.
- ***ProjectName*** The name assigned to the project file and the name in Solution Explorer where it has been copied. Don't attach the extension of the project type to this argument.
- ***Exclusive*** If this parameter is set to *true*, the current solution is closed and a new one created before the template project is added. If this parameter is *false*, the solution isn't closed and the newly created project is added to the currently open solution.

> **Note** If the *Exclusive* parameter is set to *true* when *AddFromFile* or *AddFromTemplate* is called, the existing project is closed without the user being given the option to save any modified files. You should give the user the option to save by calling the *ItemOperations.PromptToSave* property before calling *AddFromTemplate* or *AddFromFile*.

AddFromTemplate and *AddFromFile* will add a template project from anywhere on disk; that is, the files don't need to be stored in a specific location—just a place that is convenient to find. A common place to store the template files is in a folder named Templates that has been placed in the same folder as the COM object implementing the wizard. If the wizard is built using a language supported by the .NET Framework, you can use reflection to calculate the path to the templates using code like this:

```
string templatePath =
    System.Reflection.Assembly.GetExecutingAssembly().Location;
templatePath = System.IO.Path.GetDirectoryName(templatePath) +
    "\\Templates\\";
```

The *AddFromTemplate* method adds a project template to a solution, but the *ProjectItems* collection has a series of methods for adding files to an existing project: *AddFromDirectory*, *AddFromFileCopy*, *AddFromFile*, and *AddFromTemplate*. *AddFromDirectory* accepts as a parameter the path to a folder on disk; this folder is searched recursively, causing all its contained files and subfolders to be added to the project. *AddFromFileCopy* and *AddFromFile* both perform the same basic operation, adding a reference to the specified file on disk to the project. However, *AddFromFileCopy* copies the file into the project's directory structure before adding this reference. *AddFromFileCopy* differs from the *AddFromTemplate* method of the *ProjectItems* collection (not to be confused with the *AddFromTemplate* method of the *Solution* object) in that *AddFromTemplate* copies the file into the folder on disk for the project and then the project might make some modifications to the file after the files are added.

Here are the signatures and parameters for these methods:

```
public EnvDTE.ProjectItem AddFromDirectory(string Directory)
public EnvDTE.ProjectItem AddFromFileCopy(string FilePath)
public EnvDTE.ProjectItem AddFromFile(string FileName)
public EnvDTE.ProjectItem AddFromTemplate(string FileName, string Name)
```

- ***Directory*** The source folder on disk. Searches for files and subfolders begin with this folder.

- ***FilePath / FileName*** The location of the file to copy or add a reference to.
- ***Name*** The resulting name of the file. This name should have the extension of the file type.

Each of these methods returns a *ProjectItem*, an object that can be used to perform operations on the file that was added (such as opening the file or accessing the file's contents).

Solution Filenames and the New Project Wizard

When a New Project wizard is run, a solution filename might or might not be specified within the *ContextParams* array, depending on whether the user has selected the Create Directory For Solution check box, which is visible after the user clicks More in the New Project dialog box. If the check box is selected, the New Solution Name box is enabled, allowing the user to specify a new directory name for the solution. If the user doesn't select the check box, when a project is created using *Solution.AddFromTemplate* you should use the name specified for the project in the *ContextParams* array as the name of the project, the name of the solution file (if the exclusive argument in the *ContextParams* array is *true* and a solution is not currently open), and the name of the folder on disk to contain those files. These solution and project files should also be stored in the same folder. If the user selects the check box, the solution name argument in the list of context parameters is valid and you should name the root directory for the solution and the solution file using the solution name argument.

To create and name a solution file, you can use the *Solution.Create* method (as discussed in Chapter 8) by passing in the path for where to store the solution file and the name of the solution as arguments. Under the directory for the solution file, you should create a new folder to contain the project file, and you should name both the folder and the project with the project name passed into the *ContextParams* array.

Replacements

When you use a template to create a new project or a new file, the code that's generated will most likely not match the requirements for your wizard. For example, if the C# Class Library Wizard is run, the class that is generated is named *Class1*. The user can modify this class manually to give it a different name, but it's better to dynamically give the class a name that reflects the kind of class the wizard is generating (such as the name *Wizard* if the class implements a wizard). You can do this by replacing specific textual tokens within the template files after they've been added to the solution or project. To make a

replacement, you use the editor object model to search for the token, and then you modify the token's text. Tokens can be just about any text that is placed in the file, but normally they have a specific format that is distinguished from other text within the file. A common token used as a placeholder for the class name is *%CLASSNAME%*. The template for the class with the tokens added would look something like this:

```
public class %CLASSNAME%
{
    public %CLASSNAME%()
    {
        //
        // TODO: Add constructor logic here
        //
    }
}
```

The following macro, named *MakeReplacements*, replaces tokens in a file. Some of the concepts that this macro uses, such as the *EnvDTE.TextPoint* objects, might be unfamiliar to you, but we'll cover them in Chapter 11.

```
Sub MakeReplacements(ByVal projectItem As EnvDTE.ProjectItem, _
                     ByVal token As String, _
                     ByVal replaceWith As String)
    Dim window As EnvDTE.Window
    Dim textDocument As EnvDTE.TextDocument
    Dim textRanges As EnvDTE.TextRanges
    Dim findOptions As Integer
    findOptions = EnvDTE.vsFindOptions.vsFindOptionsFromStart + _
        EnvDTE.vsFindOptions.vsFindOptionsMatchCase + _
        EnvDTE.vsFindOptions.vsFindOptionsMatchWholeWord

    'Open the specified project item.
    ' This will open the file but show it hidden:
    window = projectItem.Open(EnvDTE.Constants.vsViewKindTextView)

    'Find the TextDocument object for the project item:
    textDocument = window.Document.Object("TextDocument")

    'Replace all the text that matches token with the replaceWith text:
    textDocument.ReplacePattern(token, replaceWith, _
                                findOptions, textRanges)
End Sub
```

Once you've opened the file that contains the class definition and made it the active document (by using the *ProjectItem* object returned by the *Add** methods of the *ProjectItems* collection), you can call the following macro to replace the *%CLASSNAME%* token with the class name *MyClass*:

```
Sub MakeReplacements()
    MakeReplacements(DTE.ActiveWindow.ProjectItem, _
        "%CLASSNAME%", "MyClass")
End Sub
```

Among the many variations on searching for tokens and replacing the text is deleting the text between two separate tokens. This technique is useful if the user selects an option in the user interface of a wizard that would cause a bit of code not to be needed. The Add-in Wizard uses this technique to remove the code for creating a command bar button when the Yes, Create A 'Tools' Menu Item check box on the Choose Add-in Options page of the Add-in Wizard has been cleared. The following macro deletes the text between two tokens:

```
Sub DeleteBetweenTokens(ByVal projectItem As EnvDTE.ProjectItem, _
                        ByVal token1 As String, _
                        ByVal token2 As String)
    Dim window As EnvDTE.Window
    Dim textDocument As EnvDTE.TextDocument
    Dim tokenEndPoint As EditPoint
    Dim tokenStartPoint As EditPoint
    Dim findOptions As Integer
    findOptions = EnvDTE.vsFindOptions.vsFindOptionsMatchCase + _
                  EnvDTE.vsFindOptions.vsFindOptionsMatchWholeWord

    'Open the specified project item.
    ' This will open the file, but show it hidden:
    window = projectItem.Open(EnvDTE.Constants.vsViewKindTextView)

    'Find the TextDocument object for the project item:
    textDocument = window.Document.Object("TextDocument")

    'Create edit points for searching:
    tokenEndPoint = textDocument.StartPoint.CreateEditPoint()
    tokenStartPoint = textDocument.StartPoint.CreateEditPoint()

    'Loop while all start / end tokens can be found:
    While (tokenStartPoint.FindPattern(token1, findOptions))
        If (tokenEndPoint.FindPattern(token2, findOptions, tokenEndPoint)) _
            Then

            'Move the selection to bracket the start / end tokens:
            textDocument.Selection.MoveToPoint(tokenStartPoint, False)
            textDocument.Selection.MoveToPoint(tokenEndPoint, True)

            'Delete the selection:
            textDocument.Selection.Delete()
        Else
```

```
            Exit While
        End If
    End While
End Sub
```

If our template code were modified to look like this

```
public class %CLASSNAME%
{
    public %CLASSNAME%()
    {
        //
        // TODO: Add constructor logic here
        //
    }

%BEGINOPTIONALCODE%
    void SomeOptionalCode()
    {
    }
%ENDOPTIONALCODE%
}
```

after running this macro

```
Sub MakeReplacements2()
    MakeReplacements(DTE.ActiveWindow.ProjectItem, _
        "%CLASSNAME%", "MyClass")
    DeleteBetweenTokens(DTE.ActiveWindow.ProjectItem, _
                        "%BEGINOPTIONALCODE%", "%ENDOPTIONALCODE%")
End Sub
```

the following code would result:

```
public class MyClass
{
    public MyClass()
    {
        //
        // TODO: Add constructor logic here
        //
    }
}
```

Raw Add New Item Templates

An Add New Item wizard is generally used to add a file to a project and then modify the file by making replacements to it. However, sometimes a template file doesn't need to be modified after it's been inserted into a project. An

example of this is a text file. When the user chooses to add a text file to a project, a blank file is added. Creating a wizard object just to insert a blank file is a waste of both disk space (to hold the wizard DLL) and time. To get around this, Visual Studio .NET allows what are called *raw templates*. When displaying the Add New Item dialog box, Visual Studio .NET not only searches for and shows .vsz files in the right panel of that dialog box but it also shows any other files within the folder where .vsz files can be placed. If the user selects one of these raw template files in the Add New Item dialog box, the equivalent of an *AddFromFileCopy* is performed on the file—the file is copied into the directory structure for the project that the item is being added to, and then the file is added to the project. To create a raw template, you simply create a file and place that file into the path returned by calling the *ProjectItemsTemplatePath* method and specifying the appropriate project type.

Custom Wizards

Visual Studio .NET has built-in support for creating only two types of wizards, New Project and Add New Item wizards. However, at times you might need to build a wizard that doesn't fit either of these types. Visual Studio .NET supports an extensible wizard architecture that allows you to create and invoke your own type of wizard, called a Custom wizard. An example of a Custom wizard is a wizard you can invoke to insert common code constructs, such as default implementations of classes, methods, or properties, directly into an existing source code file.

Why Custom Wizards?

The Visual Studio .NET automation group didn't add the ability to create Custom wizards simply to provide another way to extend a program with your own software creations; Custom wizards were born out of necessity. The Visual C++ group needed a way to add new functions and variables to classes from within the Class View tool window, and they wanted to do it in a wizard-like way. To make this possible, they added Custom wizards to the list of wizard types. They created wizards to add both functions and variables, and by calling the *DTE.LaunchWizard* method with the proper parameters to programmatically launch the wizards, they were able to allow the user to modify a class in a way that is familiar to them. So when you right-click on a C++ class within the Class View window and choose Add | Add Function or Add | Add Variable, you're really running a Custom wizard.

To create a Custom wizard, you build a COM object that implements the *IDTWizard* interface and you create a .vsz file for that wizard just as you would for a New Project or Add New Item wizard. However, unlike with the other types of wizards, you select the list of context parameters that your wizard takes, and a Custom wizard is invoked through your own code rather than through a dialog box. As you've seen, a list of context arguments is passed to a wizard when it is run, supplying information about how the wizard should do its work. If a wizard is to be run as a New Project or Add New Item wizard, Visual Studio .NET calculates the proper context parameters array and passes that array to the *IDTWizard.Execute* method. A Custom wizard is started programmatically, either from an add-in, a macro, or another wizard, and the calling application fills in the context parameters array.

> **Note** There are no restrictions on the context parameters that can be passed to a Custom wizard, but we recommend that you make the first argument a GUID that the caller and the wizard agree on beforehand. The wizard should then verify that the GUID passed is the expected GUID before proceeding. This approach helps keep the wizard from incorrectly using the context parameters and throwing an exception or crashing.

Running a Custom Wizard Programmatically

Your program could manually load a wizard COM object, find the *IDTWizard* interface of that wizard, and pass off the appropriate values to the *Execute* method, but it would be easier to let Visual Studio .NET handle much of this work for you. You can use the automation model to launch wizards programmatically using the *DTE.LaunchWizard* method. This method takes as its arguments the path to a .vsz file and an array of context parameters. Because the path to the .vsz file is passed to this method, the .vsz file for a Custom wizard can be stored anywhere on disk—it doesn't have to be in a specific location, as the other wizard types do. Once called, the *LaunchWizard* method instantiates the wizard COM object defined within the .vsz file and creates the custom parameters array that the .vsz file contains. The *LaunchWizard* method then passes off all the necessary values to the *Execute* method of that wizard.

You can see the *LaunchWizard* method in use in the sample CustomWizard. This wizard first verifies that the first argument of the *ContextParams* array is the expected wizard type GUID, and then it simply walks the list of context

and custom arguments that it is passed, displaying a message box for each item that it finds in those arrays. Here's the macro that starts this wizard running:

```
Sub CallCustomWizard()
    Dim contextParams(1) As Object
    contextParams(0) = "{9A4B2CFF-7A69-4671-BFA5-AE0D0C44AEFB}"
    contextParams(1) = "Hello world!"
    DTE.LaunchWizard("C:\samples\CustomWizard.vsz", contextParams)
End Sub
```

If the wizard sample is placed in the folder C:\samples, this macro packs the wizard type and the string *"Hello world!"* into an array of type object and then calls *LaunchWizard*. The wizard defined the GUID {9A4B2CFF-7A69-4671-BFA5-AE0D0C44AEFB} and expects this string as the first element of the *ContextParams* array; if the wizard doesn't get this string, it will refuse to run and will return immediately with an error.

Chaining Custom Wizards

You can use the *LaunchWizard* method to chain wizards together, which means calling one wizard within another wizard to simplify creating a project. Suppose you need a solution that contains an XML Web service and you need a Windows Forms application to gather data from that XML Web service and display it to the user. Creating the form project is simple enough: you run the Windows Form Wizard to create the Windows Forms template project, and then you create the wizard object that will add the form template to a solution. But creating the XML Web service for the solution isn't as easy as creating a template and adding it to the solution. An XML Web service project, once created, is found only on a Web server. None of the files for that XML Web service except the project file are found on the local computer—the project file simply points to the server and the location on the server where the files can be found. A wizard could talk to the Web server through a protocol such as Front Page server extensions, giving it the proper commands to store the files of an XML Web service, but it would be easier to call on the Web Service Wizard to create the and store those files for you.

The sample project ChainWizard, which is included with the sample files for this book, demonstrates how to do this. Here's a portion of the *Execute* method of this wizard:

```
const string serviceName = "ChainWizardWebService";
object []contextParamsChain = new object[7];
EnvDTE.wizardResult wizardResultChain;
EnvDTE.DTE dte = (EnvDTE.DTE)Application;

// Add our web service by filling in the context parameters,
// and chain to the Web Service Wizard
```

```
contextParamsChain[0] = EnvDTE.Constants.vsWizardNewProject;
contextParamsChain[1] = serviceName;
contextParamsChain[2] = "http://localhost/" + serviceName;
contextParamsChain[3] = System.IO.Path.GetDirectoryName(dte.FullName);
contextParamsChain[4] = (bool)ContextParams[4];
contextParamsChain[5] = "";
contextParamsChain[6] = false;
string webSvcTemplatePath = dte.Solution.get_TemplatePath(
    VSLangProj.PrjKind.prjKindCSharpProject);
webSvcTemplatePath += "CSharpWebService.vsz";
wizardResultChain = dte.LaunchWizard(webSvcTemplatePath,
    ref contextParamsChain);
```

This code creates and fills in the context parameters that are sent to the wizard object, which is run with the call to *LaunchWizard*. You calculate the location of the .vsz file for the XML Web service by using the *TemplatePath* property and passing the project type for the C# project language. Note that this code makes no attempt to create a unique XML Web service name every time it is run. If the wizard is run multiple times, you should either delete the XML Web service created on the server or change the variable value *serviceName* to a unique value.

The result of running the ChainWizard sample is a solution containing an XML Web service project and a Windows Forms project that consumes the functionality of the XML Web service.

Lab: Decoding Wizard Parameters

How do you determine which arguments to pass to a wizard during chaining? You can do it by tricking Visual Studio .NET into calling a throwaway wizard whose sole use is to capture and let you debug the custom and context parameters passed to the wizard. This is what I did to find what should be passed to the ASP.NET Web Service Wizard in the ChainWizard sample.

To start, run the Visual Basic .NET or C# Class Library wizard from the New Project dialog box. Then modify that project to implement the *IDTWizard* interface and register the library as a COM object by assigning the code a GUID and a ProgID and setting the flag in the project Property Pages dialog box to register as a COM object, just as you would for other wizards. The next step is to change the .vsz file to point to this throwaway wizard; because our example chains to the C# Web Service Wizard, we'll modify the .vsz file for that wizard. Search in the Visual Studio .NET 2003\VC#\CSharpProjects folder for the file named CSharpWebService.vsz and open the file in Notepad. We're about to modify this file, so it might be a good idea to make a backup copy.

With this .vsz file open in Notepad, simply change the ProgID from VsWizard.VsWizardEngine.7.1 to the ProgID of the throwaway wizard, and then save the .vsz file. Now, back in Visual Studio .NET, where you have the wizard project open, place a breakpoint on the *Execute* method of your wizard and press F5. (You'll need to set Visual Studio .NET, or devenv.exe, as the debug target first.) In the new instance of Visual Studio .NET that appears, open the New Project dialog box and run the C# ASP .NET Web Service Wizard. Your wizard should be called in place of the ASP.NET Web Service Wizard, and when the breakpoint on the *Execute* method is hit, you can spy on what kind of data is passed to the wizard through the custom and context parameters.

Don't forget to restore the correct .vsz file—otherwise, the throwaway wizard will be run when you actually want to create an XML Web service.

The Wizard Helper Library

As you've seen, creating wizards isn't a very complicated task. By simply creating a COM object that implements the *IDTWizard* interface and placing a .vsz file on disk, you can create a wizard that the user can run by using the New Project or Add New Item dialog box. But creating and displaying the user interface for a wizard can be tedious, which is why we've avoided the topic of wizard user interfaces until now. To create the user interface for a wizard, you must create a Windows Form and the pages for the wizard. The Windows Form will display the pages of the wizard, and the Next, Back, Finish, and Cancel buttons must properly navigate between these pages.

Much of the code to display the user interface for a wizard is boilerplate code and is similar for all wizards. To make creating wizards with a user interface easier, we've included in the book's sample files the source code for a library that manages this user interface. Simply called WizardLibrary, the library implements the *IDTWizard* interface and also handles splitting the *ContextParams* array into separate variables, making wizard creation less error-prone. To use this library to implement a wizard, you simply create a user control for each page of the wizard, write a small amount of code to let the library know which pages are available, and then implement wizard-specific functionality such as creating and adding project code.

Let's use this library to build a wizard that generates the code for a wizard—a "Wizard Wizard." First, we create a C# class library project called WizardBuilder, and then we can add a reference to the WizardLibrary assembly

(you must load and build this project from the example source files first so that the library code can be referenced) and then derive the class within our project from *InsideVSNet.WizardLibrary.WizardLibrary* (the base class that implements the functionality for the library). After making the changes to register the object for COM, our code will look like this:

```
[GuidAttribute("1EF6B85C-FD5C-4fb4-BA4D-
5ED221195DBF"), ProgIdAttribute("WizardBuilder.Wizard")]
public class Wizard : InsideVSNet.WizardLibrary.WizardLibrary
{
    public Wizard()
    {
    }
}
```

With this basic startup code, we can define the pages that the wizard displays to the user. The first page contains two options that the user can modify: an option to create an Add New Item or New Project wizard and an option to specify where the user can run the wizard. These options take care of creating the .vsz file and placing it in the correct place for the wizard that WizardBuilder generates. The second page allows the user to specify how many pages the resulting wizard code has. Since the library uses .NET user controls to implement each page of our wizard, we can use the Add New Item dialog box to add two user controls—Page1 and Page2—to our project and then add the appropriate windows controls to these forms.

With the two user controls, or pages, added to our project, we need some way for the wizard library to communicate with each page to let it do the work of generating the output project and modifying the source code files that are generated. We can do this by having each page implement the interface *InsideVSNet.WizardLibrary.IWizardPage*, which is defined by the library and has this signature:

```
public interface IWizardPage
{
    void PerformWork1(WizardLibrary WizardLibrary);
    void PerformWork2(WizardLibrary WizardLibrary);
    string HeadingLabel
    {
        get;
    }
    string DescriptionLabel
    {
```

```
        get;
    }
    System.Drawing.Image Icon
    {
        get;
    }
    void ShowHelp();
    void Initialize (WizardLibrary wizardLibrary);
}
```

Here are the methods and properties of this interface:

- ***PerformWork1*** This method is called when the user clicks the Finish button and the wizard page should start generating code. Each page is responsible for generating its own code within the project, and that work is done within this method.
- ***PerformWork2*** Each wizard page has its *PerformWork1* method called in the order that the pages are displayed. However, sometimes one page's output might depend on the output of another page. Information can be generated in the *PerformWork1* method, saved, and retrieved for further processing during the *PerformWork2* method. After the *PerformWork1* method for each page has been called, *PerformWork2* is called in the display order of each page.
- ***HeadingLabel*** This read-only property allows your wizard page to return information about the headline that's displayed in the top line of the wizard. In the Add-in Wizard's user interface, the top portion, or *banner*, of the user interface displays three pieces of information to the user: a headline displayed in bold text, descriptive text, and an icon for the page. This property returns the headline for the page.
- ***DescriptionLabel*** This property returns the description string to be displayed in the wizard banner.
- ***Icon*** This property returns a picture in the format of a *System.Drawing.Image* type to display in the banner of the wizard.
- ***ShowHelp*** This method is called when the Help button in the lower left of the wizard is clicked, signaling that the user is requesting help for the page.
- ***Initialize*** This method is called after the wizard page has been added to the list of pages maintained by the library.

With this interface implemented by each user control, we can tell the wizard library about the pages. The library implements the *IDTWizard* interface and its *Execute* method for us, but when the wizard is first run, it calls a method defined in the *WizardLibrary* class (from which we derived our wizard class), which is declared as abstract and is also called *Execute*. This method, which should be placed within the class that inherits from the *WizardLibrary* class, is defined as follows:

```
public override void Execute(EnvDTE.DTE applicationObject);
```

This version of *Execute* is where we set up the wizard library to let it know which pages are available to it. You add pages by calling the *WizardLibrary.AddPage* method, passing an instance of one of our user controls that implements the *IWizardPage* interface in the order that they should appear in the user interface for your wizard. We can create and add the two user controls we created earlier (*Page1* and *Page2*) within the *Execute* method using code such as this:

```
public override void Execute(EnvDTE.DTE applicationObject)
{
    Title = "Wizard Builder";
    AddPage(new Page1());
    AddPage(new Page2());
}
```

This code not only tells the library about the pages of our wizard but also sets the title of the wizard dialog box to *Wizard Builder*. The *WizardLibrary* class defines a property, *Title*, that sets the text of the user interface form for the wizard. When this *Execute* method returns, the wizard library has all the information it needs to run. The library then displays the Windows Form dialog box with the user control pages displayed and implements the proper navigation between pages of the wizard. The wizard library also manages the *wizardResult* parameter of the *IDTWizard.Execute* method, returning *wizardResultSuccess* if the Finish button is clicked, *wizardResultCancel* if the Cancel button is clicked, *wizardResultFailure* if an exception is thrown, and *wizardResultBackOut* if the first page of the wizard is displayed and the user clicks the Back button. With this code written, when the wizard is run, the dialog box in Figure 9-3 is shown with the first page of the wizard displayed.

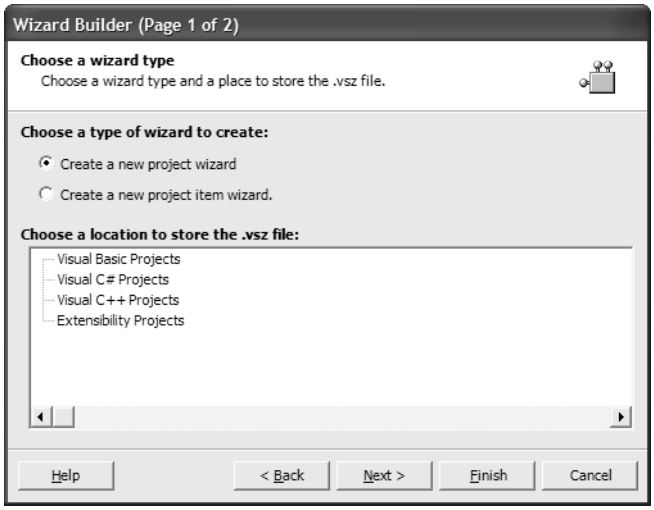

Figure 9-3 The first page of the WizardBuilder sample, with the first page displayed

Wizard Variables

As we've discussed, when the *IDTWizard.Execute* method is called, information is passed to the wizard through the *ContextParams* and *CustomParams* arguments. But the *Execute* method that your wizard implements isn't passed these arguments because the wizard library handles extracting the variables from the *ContextParams* array and storing them as member variables. The values of *CustomParams* aren't extracted in the same way as *ContextParams* because these variables are specific to your wizard and the library has no previous knowledge about what it contains. Table 9-3 and Table 9-4 list the variable names you can use, how they correspond to the values listed in Table 9-1 and Table 9-2, and the context in which you can use them.

Table 9-3 New Project Wizard Library Variables and Their Corresponding *ContextAttribute* Array Values

Variable	Corresponding Value
wizardType	Wizard Type
newProjectName	Project Name
newProjectLocation	Local Directory

(continued)

Table 9-3 New Project Wizard Library Variables and Their Corresponding *ContextAttribute* Array Values *(continued)*

Variable	Corresponding Value
visualStudioInstallDirectory	Installation Directory
exclusiveProject	Exclusive
newSolutionName	Solution Name
runSilent	Silent

Table 9-4 Add Item Wizard Library Variables and Their Corresponding *ContextAttribute* Array Values

Variable	Corresponding Value
wizardType	Wizard Type
projectName	Project Name
projectItems	Project Items
newItemLocation	New Item Location
newItemName	New Item Name
productInstallDirectory	Product Install Directory
runSilent	Silent

The variables listed in Table 9-3 and Table 9-4 are generated by extracting values from the context parameters array. Two other variables are available within the wizard library, and they are available to either New Project or Add New Item wizards:

- ***application*** The DTE object for the instance of Visual Studio .NET in which the wizard is running
- ***CustomArguments*** A list of the custom parameters, copied verbatim from the *CustomParam* arguments passed to the *IDTWizard.Exec* method

The wizard library provides one other variable that your wizard can use. We mentioned earlier that the *IWizardPage.PerformWork1* method can save information for later use in the *IWizardPage.PerformWork2* method. However, one page of a wizard doesn't have access to the data of another page because they are separate objects. For storing information, the wizard library contains a data member named *customData* that has the type *System.Collections.Specialized.ListDictionary*. This data member allows one page of the wizard to store a

name and value pair for use by another page of the wizard. The sample wizard we're building here uses the *customData* member value to store information such as the *EnvDTE.Project* object, which was created with the call to *CreateProject* within the *PerformWork1* method of the first page of the wizard.

Wizard Helper Methods

The wizard library supports four helper methods that a wizard can use when generating the resulting project or file. You can use *CreateProject*, which has the following signature, to create a project based on a project template.

```
public EnvDTE.Project CreateProject(string templatePath)
```

This method creates a solution file if one is needed and places the project file in the correct folder on disk if the user specified creating separate folders for the project and solution files. The only parameter this project accepts is the path to the template project file. Values such as the name of the project and solution, as well as whether the solution file should be closed or the new project should be added to the currently open solution file, don't need to be passed to this method because these values are already known to the wizard. The final two methods, *DeleteBetweenTokens* and *MakeReplacements*, are C# versions of the macros of the same name shown earlier in this chapter; you can use them to modify the source files you create.

Completing the WizardBuilder Sample

Now that we've covered the techniques for building wizards using the library, we can complete the WizardBuilder sample. We've already created a class that derives from the *WizardLibrary* class, created the *Execute* method, added two pages in the form of user controls to the project, and implemented the *IWizardPage* interface in each of these pages. All that's left to do is to generate the output code. We'll start by creating the template files. We'll create a C# class library called WizardTemplate, specify the project setting to register as a COM object, and modify the code for the class to the following:

```
namespace %NAMESPACE%
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
     [GuidAttribute("%GUID%"), ProgIdAttribute("%NAMESPACE%.Wizard")]
    public class Wizard : InsideVSNet.WizardLibrary.WizardLibrary
    {
        public Wizard()
        {
```

```
            //
            // TODO: Add constructor logic here
            //
        }

        public override void Execute(EnvDTE.DTE application)
        {
            Title = "My Wizard";
%WIZARDPAGES%
        }
    }
}
```

When run, the wizard replaces *%NAMESPACE%* with the name of the project specified in the New Project dialog box, and *%GUID%* is replaced with a new GUID. The token *%WIZARDPAGES%* is replaced with code generated to add an instance of the pages to the library. The next template to create is the template for the pages of the wizard. We'll do this by running the C# Windows Control Library Wizard, modifying the generated user control to implement *IWizardPage*, and changing the namespace and all instances of the class name with the tokens *%NAMESPACE%* and *%PAGENAME%*, respectively. We'll copy the file for the user control into the templates folder for our wizard.

The last step is to fill out the *PerformWork1* and *PerformWork2* methods for the two pages of the wizard. *PerformWork1* for the first page handles creating the project and making replacements within the file, as outlined earlier. *PerformWork2* for this first page handles building the .vsz file, placing a copy in the folder the user specified, and adding a reference to the WizardLibrary.dll assembly. *PerformWork1* for the second page isn't used; *PerformWork2* for this page handles adding one copy of the user control template for each of the number of pages the user specified while running the wizard, and then makes the proper replacements in the newly added page. Finally, the *PerformWork2* method sets up the code in the wizard.cs file to make the replacement to the *%WIZARDPAGES%* token.

Looking Ahead

In this chapter and the previous one, we have dealt with projects and their items—manipulating them through the object model and creating them using wizards. Next we'll move on to something a little different: windows within Visual Studio .NET and how to program them.