THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES

PRACTICAL CODE GENERATION IN .NET

COVERING VISUAL STUDIO® 2005, 2008, AND 2010

PETER VOGEL

Foreword by **GARETH JONES,**
Developer Architect, Visual Studio, Microsoft

Library of Congress Cataloging-in-Publication Data:

Vogel, Peter, 1953-
Practical code generation in .NET : covering Visual Studio 2005, 2008, and 2010 / Peter Vogel.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-60678-5 (pbk. : alk. paper)
1. Microsoft Visual studio. 2. Code generators. 3. Microsoft .NET Framework. I. Title.
QA76.76.G46V65 2010
006.7'882--dc22

2010003301

ISBN-13: 978-0-321-60678-5

ISBN-10: 0-321-60678-7

Text printed in the United States on recycled paper at RR Donnelly in Crawfordsville, Indiana.

First printing April 2010

FOREWORD

"I believe raising the level of abstraction is fundamental in all practical intellectual endeavors."

—Bjarne Stroustrup, 2004

The story of software engineering has been the story of increasing the level of abstraction at which we as programmers work, from logic encoded in hardware to toggle switches representing binary digits, through machine code, assembly language, low-level languages, and high-level languages both procedural and functional. More recently, we have declarative models of business processes that can be shared and discussed with folks who have no formal training in computer science at all. I'd wager that most readers were nodding along with my list above until I got to the last item. Has abstract modeling become a proven mainstream technique yet, accepted and used by all in the industry? No, of course not; this is the abstraction increase that we're currently involved in working out, and doubts and skepticism still abound. It's hard to remember now, but all earlier progressions were surrounded by doubt as to their value as well. In the infancy of each new technique, programmers wanted detailed access to the previous layer, not fully trusting the new tool to meet their needs, but as tools and understanding matured, this requirement slipped away. Today, few developers feel the need to examine the IL or bytecode produced by their C# or Java compiler, and fewer still the assembly code produced by the JIT compiler underlying their runtime.

Code generators bridge the gap from nascent abstractions to their well understood predecessor technologies. They facilitate working on a problem at a higher, more productive level and translate that to a practical solution based on best practices at a lower level. Of course, all this talk of raising abstraction levels implies some sort of grandiose vision of defining your application with metadata and generating the whole thing. This book demonstrates clearly that nothing is further from the truth and that starting small is where the value is at when it comes to code generators.

As our abstractions get closer to our problem space, as opposed to being merely refinements in the solution space, we're inevitably going to use more granular, more fragmented tools that are specialized for the task at hand. Alongside a gradual easing of the difficulty of building such tools, we have the ingredients for a productivity explosion, driven by the creation of small tools to help with the distinct tasks in our day-to-day jobs. Unix command-line developers have known this truth since the 1980s with their chaining of small scripts; now it's becoming a reality for the IDE generation.

Whether your design-time metadata is a simple list of settings and their default values in Visual Studio, or the set of tables and stored procedures to be accessed in your relational database, there's ample opportunity for using that data to generate reliable code that conforms to proven patterns. Look for metadata that's already present in your application design, and surface it to drive tools. Look for repetitive patterns in your code and determine which pieces are fixed and where the variability is. Make creating tools and increasing task repeatability part of your normal approach to problem solving to ease your working life.

Once these skills are within your everyday comfort zone, your productivity will get a boost and your value to your team will increase. Spreading the use of such tools to make your peers more productive is an important step in the transition of our industry from one dominated by software artisans to one driven by engineering practices that provide predictable results at scale.

I encourage you to add the techniques outlined in this book to your toolset and to use them to develop your own workbench of generative tools. In doing so, I'm confident you'll improve your capabilities, and what's more, have fun doing so.

—Gareth Jones,
Developer Architect, Visual Studio
Issaquah, WA
March 2010

PREFACE

Whenever you're looking at buying a book, it seems to me there's only one question that should be asked: "Why should I invest my hard-earned money in this book?"

This book is designed to make you, as a software developer, more productive. It does that by giving you all the tools you need to incorporate code generation into your standard development practices. Why would you buy this book? Because letting Visual Studio and .NET write the boring code lets you work on the important stuff.

All the code-generation tools you need are already available to you because you're already using code generation. As soon as you start working in .NET and any version of Visual Studio, an enormous amount of code is being created for you. For instance, if you've ever created a DataSet then you've been using a Visual Studio custom tool that generates the code class file for your DataSet—and that's just some of the code that's easy to see. There's a great deal more generated code hidden away where you can't find it. In addition to making you more productive, those code-generation tools have taken over creating some of the repetitious and error-prone parts of building applications, thereby also increasing the quality of your code. The next step is for you to start using those tools to create solutions that you—rather than the .NET team—want solved.

But the problem is that there is no single point of reference for this material. Code generation requires several tools, and there is no one place where all those tools are discussed. And, even when you find resources for those tools, a comprehensive reference that shows you how to apply them is missing. So part of the answer to the question "Should I buy this book?" is that the book provides "one-stop shopping" for all the tools you need to implement code generation. I've put all the tools in one book and covered all the parts of each tool that are relevant to code generation.

I've been building Visual Studio add-ins that created code for me since .NET 1.1. As I worked with various clients, I found that they were also adopting code-generation solutions—and I got to help them do it. So, in this

book, I wanted to show how those tools could be used in a practical way—and how they would work together. As a result, almost a third of this book is taken up with three case studies that show how to coordinate these tools to create useful, reliable code-generation solutions for common problems.

To put it another way: I wrote this book because I believe that the code-generation tools built into .NET and Visual Studio 2005/2008/2010 will make you a better, more productive developer. And I believe that more developers would develop more code-generation solutions if the tools were more accessible to them.

And, of course, I wrote this book because it's cool technology. Several years ago, I stumbled across a great quote from Dick Sites (one of the designers of Digital Equipment Corporation's Alpha architecture): "I'd rather write programs that help me write programs than write programs." That seemed right.

One caveat: If you're looking for a book that shows you how to create enormous frameworks that will generate thousands of lines of code from a single XML document that describes an application—this is not that book. Certainly, all the tools you'll need are in here, but that's not my focus. I don't want to describe how to spend three years building your very own "application generator." Instead, I want to give you the tools that will let you solve a problem in your life and do it in a morning—a solution that you'll never have to think about again because it will just work.

The first case study in this book (in Chapter 9) is a good example: This solution generates a class that simplifies access to the connection strings specified in the `connectionStrings` element. It took me about a morning to write, it works in every application, it reduces the amount of code I write, and it eliminates errors in my applications. The second case study (Chapter 10) is similarly focused: It generates the code for an ASP.NET validator that checks that data entered by the user is a valid entry in a table. Like my connection string generator, this is something I use in almost every ASP.NET application I write. The final case study (Chapter 11) took a full day to write, but it allows nonprogrammers to use a visual designer to generate the code necessary to integrate their software into one of my applications—and to do it reliably and without requiring my intervention.

Here's a breakdown of the topics covered in this book:

- Chapter 1, "Introducing Code Generation," is the "theory and practice" chapter. It discusses the structure of code-generation solutions and covers best practices in architecting solutions.

- Chapter 2, "Integrating with Visual Studio," gives you enough information about creating Visual Studio add-ins for you to integrate code generation into your standard activities. The connection string generator, for instance, generates code whenever the developer closes the web.config file; the validator example generates code whenever the developer closes an .aspx file containing a specific tag.
- Chapter 3, "Manipulating Project Components," covers the objects and methods that you need to add (or remove) components to a project: code files, folders, and so on.
- Chapter 4, "Modifying Code in the Editor," gives you the tools you need to insert text into files. This allows you to generate code using any tool you want (even standard string-handling functions) and then insert that code into a file in your project.
- Chapter 5, "Supporting Project-Specific Features," provides support for working with specific types of projects: C#, Visual Basic, and ASP.NET websites. Each of these project types have special features that aren't available through the objects covered in Chapter 4.
- Chapter 6, "Generating Language-Neutral Code," contains full coverage of the CodeDom, which allows you to generate code without having to commit to producing Visual Basic or C# until it's time to insert text into files.
- Chapter 7, "Generating Code from Templates with T4," covers a new technology in .NET: Text Template Transformation Toolkit (T4), which uses a template-based approach to code generation that reduces the amount of code required in a solution.
- Chapter 8, "Other Tools: Templates, Attributes, and Custom Tools," has three technologies you can use in creating code-generation solutions: Visual Studio templates, attributes, and custom tools. Visual Studio templates reduce the code that must be generated from your code; attributes provide a way for developers to insert information into a file to specify the code to be generated; custom tools are standalone programs that, when associated with a file, read the file's contents and create a file of generated code.
- Chapter 9, "Case Study: Generating a Connection String Manager," Chapter 10, "Case Study: Generating Validation Code," and Chapter 11, "Case Study: Generating Data-Conversion Code," are the three case studies included in this book.

When I wrote this book, I assumed that you're an experienced developer with a solid command of your programming language. I also assumed that you have several years of experience in creating complex applications.

You can find code samples for this book and all three of the case studies on www.informit.com and on my website at www.phvis.com.

I hope you find this book useful and enjoyable to read.

—Peter Vogel
March 2010

CHAPTER 9

CASE STUDY: GENERATING A CONNECTION STRING MANAGER

In this chapter:

- Defining the Problem
- Setting Up the Add-In
- Creating the Code Generator
- Customizing the Template
- Generating Code
- Reading Input
- Notifying the Developer
- Supporting Customization
- Tying Generation to Events
- Generating a Simple Class

In this chapter, I walk through an end-to-end solution for code generation that concentrates on integrating with Visual Studio and working with the `CodeElement` objects. The code for this solution is kept purposely simple to avoid involving other tools. (For example, I only make minimal use of the code editor object.) The case study in the next chapter includes a wider range of tools, including the CodeDom.

I've also assumed that there will be only one configuration file open at a time—because you can only have one app.config or web.config in a project, that's not an unreasonable assumption. However, because a Visual Studio solution can include multiple projects, it's at least conceivable that a developer could have two or more configuration files open at a time. The case study in the next chapter shows a more sophisticated process for handling events to support scenarios where multiple files that trigger code generation could be open.

This solution does demonstrate how to do the following:

- Support all project types, including ASP.NET websites, without using the `VsWebsite` objects (or, at least, only having to use them once)
- Support customization by the developer
- Read existing files in the project to get the input specifications
- Create a page in the Tools | Options dialog to allow the developer to configure code generation
- Tie code generation to events in Visual Studio

As part of this solution, I include some utilities that you can use in other code-generation solutions. One caveat: To simplify the code in this example, I assume that I'm only generating C# code, although I discuss where the solution would be different when supporting Visual Basic.

Finally, within those self-imposed limitations, I've tried to demonstrate a variety of techniques to show the range of options available to you when generating code. My goal for this chapter is to demonstrate a process for developing an add-in, along with some of my best practices and design patterns I follow.

Defining the Problem

The problem I want to address is relatively simple: handling the connection strings in an application's configuration file. A typical example of the separate section available for holding connection strings in an app.config or web.config file looks like this:

```
<connectionStrings>
  <add name="Northwind" connectionString="..."
       providerName="System.Data.SqlClient" />
 </connectionStrings>
```

When retrieving the connection string, you access the connection strings as named members of a collection. To retrieve the connection string from the previous example in a non-ASP.NET application, you'd use this code:

```
string MyConnection = ConfigurationManager.
              ConnectionStrings["Northwnd"].ConnectionString;
```

Here is the syntax for an ASP.NET application:

```
return System.Web.Configuration.WebConfigurationManager.
              ConnectionStrings["Northwnd"].ConnectionString;
```

This syntax creates problems for developers. The absence of IntelliSense support when specifying the connection string means that you have to switch back to your configuration file in order to find what connection strings you have and what you called them; if you mistype the name of the connection string, you won't find that mistake until that line of code executes (probably when someone who has input into your job appraisal is looking over your shoulder). You don't have to take my word that this syntax is error-prone: Did you catch the misspelling of "Northwnd" in the sample code? It should have been "Northwind" to match the `connectionString` example—but if you don't spot that problem when reading the code, you won't find it until the code executes.

A Model Solution

ASP.NET provides a better model for handling connection strings in the way that the personalization provider handles properties. As with connection strings, you define personalization properties by entering XML tags into your website's configuration file. A typical example looks like this:

```
<properties>
   <add name="LinesPerPage" type="int" defaultValue="0"/>
</properties>
```

Unlike connection strings, however, at runtime, you don't access your personalization properties as members of a collection. Instead, you access the properties you defined in the Web.config through properties on the Profile object, like this:

```
Profile.LinesPerPage = 15;
```

When entering this code you get full IntelliSense support for all the Profile properties (see Figure 9-1). If you ask for a property that doesn't exist, your problem is found at compile time, not runtime. Overall, personalization delivers a solution that provides better support to developers than is available with connections strings, even though the input to both processes is the same.

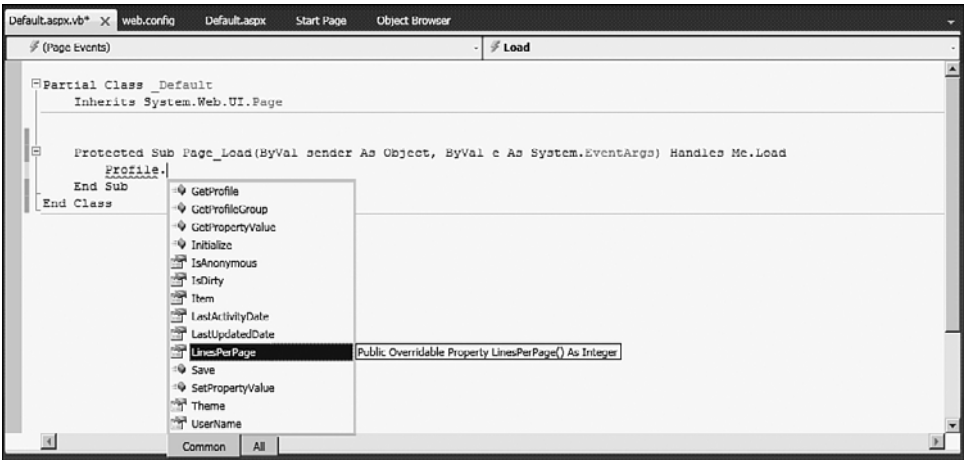

FIGURE 9-1 Although personalization properties are defined in an application's configuration file, access to those properties is handled through specific properties on the Profile object.

The personalization solution is made possible through the magic of code generation: By analyzing the entries in the configuration file, Visual Studio and ASP.NET generate a `Profile` object with all the properties specified in the Web.config file's XML tags. Because the data type for each property is specified in the XML tags, the generated code isn't "general-purpose" code with multiple `if` statements checking the data type or with all variables declared as type Object—you get the specific code you need for the properties you defined in the configuration file.

A similar solution for connection strings lets the developer write code like this:

```
string MyConnection = ConnectionManager.Northwind;
```

In this solution, the `ConnectionManager` object is a static class that isn't instantiated and has a property for each connection string. This solution gives the developer full IntelliSense support and compile-time checking when accessing a connection string.

The code for my `ConnectionManager` object looks something like this for Northwind property in a non-ASP.NET project:

```
public static partial class ConnectionManager
{
  public static string Northwind
```

```
  {
   get
   {
    return ConfigurationManager.
                        ConnectionStrings["Northwind"].ConnectionString;
   }
  }
}
```

Supporting Customization

The ConnectionManager solution also allows the developer to introduce custom code to support those instances where a full connection string isn't stored in the configuration file. For instance, one of my clients provides data gathering and storage services to their customers. To support scalability (and to help ensure privacy), each customer's data is kept in a separate database. As a result, my client stores a template connection string in the application's configuration file. The template contains replaceable components that support tailoring the connection string for each customer. In my client's application, whenever a connection string is retrieved, code in the application modifies the template and tailors the string to work with a specific customer's database. The ConnectionManager solution supports this kind of modification by allowing the developer to step in and add his or her own code to the process.

Setting Up the Add-In

Practical code-generation solutions should seamlessly integrate with the developer's normal activities. I begin this project by creating the add-in that will trigger code generation whenever the config file is closed (or whenever the developer chooses to generate code by selecting a menu option).

Defining the Add-In

I start a new code-generation project either by extending an existing add-in with similar functionality or creating a new one. For this project, I start a new add-in. I always begin with the simplest possible interface for triggering the code generation: a single menu item that runs the solution. This simplifies testing and debugging. Near the end of the project, I convert this

add-in to run when the project is built or when the configuration file is closed.

From the File menu, I select New Project and, in the New Project dialog, under Extensibility, I select the Visual Studio Add-In template. After giving my add-in a name (I used "ConnectionManager") and specifying a folder to keep it in, I click the OK button to start the wizard. In the wizard, I take the following actions:

- Select C# as the language.
- Deselect Microsoft Visual Studio Macros.
- Replace the default name and description for the add-in with my own text.
- Select all three choices on the fourth page:
  - Add a command to the Tools menu.
  - Have the add-in load with Visual Studio.
  - Promise not to put up a modal dialog.
- Add some information for the About dialog.

Once the project is created, I modify the project's properties (as described in Chapter 2, "Integrating with Visual Studio"):

- Set the assembly name (and make the same change in the `<Assembly>` element of the two .addin files).
- In the Build Events tab (C#) or on the Compile tab after clicking the Build Events button (Visual Basic), I add these two lines to the Pre-build event command line. (This code is spread over two lines to fit on the page, but the second and third line should be entered as one line in the Pre-build text box.)

```
if exist "$(TargetPath).locked" del "$(TargetPath).locked"
if exist "$(TargetPath)" if not exist "$(TargetPath).locked" move
                    "$(TargetPath)" "$(TargetPath).locked"
```

Creating the Menu

Once the project is generated, my next step is to modify the code in the Connect.cs file. One of my goals when designing the Connect.cs file is to create a version that doesn't require many changes when setting up a new add-in. To support that, in the Connect.cs file I add four fields (named `menuName`, `menuItemName`, `menuItemCaption`, and `menuTooltip`) at the

top of the class. If all that's required in an add-in is a single menu item on a menu (and that's always my start point for any solution), the only changes required are to the values of these four fields:

```
string menuName = "Tools";
string menuItemName = "ConStrGentr";
string menuItemCaption = "Generate Connection String Class";
string menuToolTip = "Create a class for managing connection strings";
```

I then replace all the code in the `OnConnection` method with the following code in Visual Studio 2005/2008, which uses my fields to find the menu specified in my four fields:

```
public void OnConnection(object application, ext_ConnectMode
                         connectMode, object addInInst,
                         ref Array custom)
{
 _applicationObject = (DTE2)application;
 _addInInstance = (AddIn)addInInst;

 if (connectMode == ext_ConnectMode.ext_cm_UISetup)
 {
  object[] contextGUIDS = new object[] { };
  Commands2 commands = (Commands2)_applicationObject.Commands;

  string FoundMenuName;

  try
  {
   System.Resources.ResourceManager resourceManager = new
     System.Resources.ResourceManager(
       _addInInstance.ProgID + ".CommandBar",
       System.Reflection.Assembly.GetExecutingAssembly());
   System.Globalization.CultureInfo cultureInfo = new
       System.Globalization.CultureInfo(_applicationObject.LocaleID);
   if (cultureInfo.TwoLetterISOLanguageName == "zh")
   {
    System.Globalization.CultureInfo parentCultureInfo =
                                              cultureInfo.Parent;
    FoundMenuName = resourceManager.GetString(
             String.Concat(parentCultureInfo.Name, menuName));
   }
   else
   {
```

```
    FoundMenuName = resourceManager.GetString(String.Concat(
                cultureInfo.TwoLetterISOLanguageName, menuName));
   }
  }
  catch (Exception e)
  {
   FoundMenuName = menuName;
  }
  if (FoundMenuName == "")
  {
   FoundMenuName = menuName;
  }
```

The equivalent code in Visual Studio 2010 looks like this:

```
CommandBar cb;
bool MainMenu = true;
string MenuBarName = "Menubar";

if (MainMenu)
{
 cb = ((CommandBars)_applicationObject.CommandBars)[MenuBarName];
 cb = ((CommandBarPopup)cb.Controls[FoundMenuName]).CommandBar;
}
else
{
 CommandBars cbs = (CommandBars)_applicationObject.CommandBars;
 cb = cbs[FoundMenuName];
}
```

In Visual Studio 2008/2010, the next code adds a new menu item (with the name specified in `menuItemName`) to the menu I just found, with the caption and tooltip specified in `menuItemCaption` and `menuToolTip`:

```
  Commands2 cmds = (Commands2)_applicationObject.Commands;
  CommandBars cbs = (CommandBars)_applicationObject.CommandBars;
  CommandBar cb = cbs[FoundMenuName];

  Command NamedCommand = null;
  try
  {
   NamedCommand = _applicationObject.Commands.Item(
                  _addInInstance.ProgID + menuItemName, 1);
  }
  catch
```

```
  {
   try
   {
    NamedCommand = cmds.AddNamedCommand2(_addInInstance,
                menuItemName, menuItemCaption, menuToolTip,
                true,50, ref contextGUIDS
               (int) vsCommandStatus.vsCommandStatusSupported +
               (int) vsCommandStatus.vsCommandStatusEnabled,
               (int) vsCommandStyle.vsCommandStylePictAndText,
              vsCommandControlType.vsCommandControlTypeButton);
   }
   catch {}

   try
   {
    CommandBarControl cbc =
                cb.Controls[menuItemCaption];
   }
   catch
   {
    NamedCommand.AddControl(cb, 1);
   }
  }
 }
}
```

In Visual Studio 2005, I would need to replace the code that uses `AddNamedCommand2` with code that uses `AddNamedCommand`:

```
Command command = cmds.AddNamedCommand(_addInInstance,
                    menuItemName,menuItemCaption, menuToolTip,
                    true,59,ref contextGUIDS,
             (int) vsCommandStatus.vsCommandStatusSupported) +
                    (int) vsCommandStatus.vsCommandStatusEnabled);
```

In addition to replacing the code in the `OnConnection` method, I also need to modify the code in the `QueryStatus` method to allow me to use any menu item created by this add-in:

```
public void QueryStatus(string commandName,
                         vsCommandStatusTextWanted neededText,
                         ref vsCommandStatus status,
                         ref object commandText)
```

```
{
 if(neededText ==
        vsCommandStatusTextWanted.vsCommandStatusTextWantedNone)
 {
  if (commandName.StartsWith(_addInInstance.Name + ".Connect"))
  {
   status = (vsCommandStatus)vsCommandStatus.
                                    vsCommandStatusSupported
            |vsCommandStatus.vsCommandStatusEnabled;
   return;
  }
 }
}
```

Calling the Solution

To make the Connect.cs class as portable as possible, I put my code-generation solution in a separate class. This means that the only change required to the `Exec` method of Connect.cs is the name of the class and method that implements the solution.

For this solution, I have the `Exec` method instantiate a class called `DatabaseUtilities` and call a method named `GenerateConnectionManager`. I pass the `DTE2` object that provides access to Visual Studio to the constructor for this code-generation class. As a result, the code in this `Exec` method to create the `DatabaseUtilities` class, pass the `_applicationObject` variable that holds the `DTE2` object, and call the `GenerateConnectionClass` method looks like this:

```
public void Exec(string commandName,
                 vsCommandExecOption executeOption,
                 ref object varIn, ref object varOut,
                 ref bool handled)
{
 handled = false;
 if(executeOption == vsCommandExecOption.
                                  vsCommandExecOptionDoDefault)
 {
  if (commandName == _addInInstance.ProgID + "." + menuItemName)
  {
   DatabaseUtiltiies dbu = new
                           DatabaseUtiltiies(_applicationObject);
   dbu.GenerateConnectionClass();
```

```
    handled = true;
    return;
   }
  }
}
```

Creating the Code Generator

With the framework for calling my code-generation solution in place, I'm ready to start creating the code-generation code in my `DatabaseUtilities` class. The constructor for the class accepts the reference to the `DTE2` object and moves it to a field in the class. The initial version of the class also contains the `GenerateConnectionManager` method that's called from my add-in:

```
using System;
using System.Collections.Generic;
using System.Text;
using EnvDTE;
using EnvDTE80;

namespace ConnectionStringGenerator
{
     class DatabaseUtiltiies
     {
          DTE2 applicationObject;

          public DatabaseUtiltiies(DTE2 ApplicationObject)
          {
               applicationObject = ApplicationObject;
          }

          public void GenerateConnectionManager()
          {

          }
     }
}
```

At this point, I've got enough code to start testing my solution by adding a line of code to my `GenerateConnectionManager` method that writes to the status bar:

```
applicationObject.DTE.StatusBar.Text = "Code generator called.";
```

I can now check that the menu item appears (with all the text spelled correctly), that I can load my generation class, and that I can successfully call the generation method. If all that works, I'm ready to start thinking about what the solution will do.

Finding the Project

The first step in this code-generation project is to retrieve a reference to the project that the developer wants to modify. At this point, it's worthwhile to think about the problem from the point of view of the developer for whom you're generating the code. When the developer clicks the menu item that starts the code-generation process, what project will the developer expect your code to work with?

For a code-generation solution run from a button on a menu, my first choice is to work with the project for the currently open document. This code retrieves the project for that document:

```
Project prj = null;
if (applicationObject.ActiveDocument != null)
{
 prj = applicationObject.ActiveDocument.
                                    ProjectItem.ContainingProject;
}
```

However, if there is no open document, my second choice is to work with the project for the item currently selected in Solution Explorer. This code checks to see if an item is selected in Solution Explorer and if the item has an associated `ProjectItem`. Then, if both of those conditions are true, it retrieves the associated `Project`:

```
else
{
 if (applicationObject.SelectedItems.Count > 0)
 {
  if (applicationObject.SelectedItems.Item(1).ProjectItem
                                                        != null)
  {
```

```
    prj = applicationObject.SelectedItems.Item(1).
                                    ProjectItem.ContainingProject;
   }
   else
   {
    prj = applicationObject.SelectedItems.Item(1).Project;
   }
  }
```

Unfortunately, there is a possibility that no document is open, that nothing is selected in Solution Explorer, or that the selected item doesn't return a `Project` reference. If I can't determine the `Project`, I give up and exit. However, the decent thing to do in that situation is to tell the developer that no code has been generated. It's tempting to pop up a form telling the developer that no code was generated, but when working through the Add-In Wizard, I promised never to display a modal dialog. So, instead I just update Visual Studio's status bar with code like this. (In the section "Notifying the Developer" later in this chapter, I enhance the messaging to use the `TaskList` for more serious messages.)

```
if (prj == null)
{
 applicationObject.DTE.StatusBar.Text =
                                    "Please select a project item.";
 return;
}
```

Assuming that I get a reference to the `Project`, I now get references to the `Project`'s `ProjectItems` collection and the `Solution` it's part of—I'll need both of these objects later in the solution:

```
ProjectItems pjis = prj.ProjectItems;
Solution2 sln = (Solution2) applicationObject.Solution;
```

Does Anything Need to be Done?

In the process I recommended in Chapter 1, "Introducing Code Generation," as part of reading your inputs you should determine whether any code needs to be generated. In this case, that means retrieving the web.config file and determining if it contains any `connectionString` elements.

This code attempts to retrieve the project's web.config file and, if that fails, the project's app.config file. If neither exists, the code exits:

```
ProjectItem cfg = null;
try
{
 cfg = pi.Project.ProjectItems.Item("web.config");
}
catch
{
 try
 {
  cfg = pi.Project.ProjectItems.Item("app.config");
 }
 catch {}
}
if (cfg == null)
{
 return;
}
```

With a configuration file found, the code loads its contents into an XML document by passing the full pathname to the configuration file to an `XmlDocument` object. The code then uses an XPath expression to search the document for `connectionString` elements. If none are found, the code exits:

```
System.Xml.XmlDocument dom;
dom = new System.Xml.XmlDocument();

dom.Load(@cfg.Properties.Item("FullPath").Value.ToString());
System.Xml.XmlNode ndCons =
      dom.SelectSingleNode("//connectionStrings");
if (ndCons == null || ndCons.ChildNodes.Count == 0)
{
 return;
}
```

Segregating Generated Code

I'm almost ready to start adding code, but I need to decide how to handle the files containing my generated code. Although many code generators attempt to hide generated classes from the developer, my preference is to leave the code visible. (Among other benefits, this makes it easier for me

to check that I'm generating the right code during development and debugging.)

However, I do segregate my generated code into special folders. Using the reference to the `ProjectItems` collection, I can add that folder to hold my generated code using the `AddFolder` method. For most projects, I create a folder called "Generated Code" to put the class file in. However, for ASP.NET projects, I place the class file in the App_Code folder.

These folders may already be present. (Even my own Generated Code folder may already exist if the developer has run this add-in, or another one of my code-generation utilities, before.) Adding the folder a second time will raise an error; however, rather than check that the folder already exists, I just catch the error and discard it. I'll need to access the folder again, so after adding it I retrieve a reference to the new folder through the `ProjectItems`' `Item` method (unfortunately, the `AddFolder` method doesn't return a reference to the new folder) and store it.

I begin by declaring a field to hold the reference to the folder with the generated code:

```
ProjectItem codeFolder;
```

In the following code, I first check to see what kind of project I have by looking at the GUID in the `Project` object's `Kind` property. If it's an ASP.NET project, I attempt to add the App_Code folder. For any other kind of project, I add a folder named "Generated Code." As I noted before, if the folders already exist, I just catch the error and discard it. After attempting to add the folder, I get a reference to it:

```
if (prj.Kind == "{E24C65DC-7377-472b-9ABA-BC803B73C61A}")
{
 try
 {
  pjis.AddFolder("App_Code",
                 "{6BB5F8EF-4483-11D3-8BCF-00C04F8EC28C}");
 }
 catch { };
 codeFolder = pjis.Item("App_Code");
}
else
{
 try
 {
  pjis.AddFolder("Generated Code",
                   Constants.vsProjectItemKindPhysicalFolder);
```

```
 }
 catch {};
 codeFolder = pjis.Item("Generated Code");
}
```

I could simplify the code required to add the App_Code folder by using the `VsWebSite` objects (described in Chapter 5, "Supporting Project-Specific Features"). However, for this case study, one of my goals is to use as few tools as possible, which means avoiding using the project-specific objects described in that chapter.

With the folder in place, I add the class file that will eventually contain my ConnectionManager code. At this point I have to decide how I want to handle regeneration when the developer is generating the code for the second (or subsequent) time. The simplest strategy for supporting regeneration is to find the file containing the code from the previous generation and delete it. The alternative is to attempt to reconcile the previously generated code against the current environment, a process that is both difficult to implement and error-prone. (One solution is demonstrated in the case study in Chapter 10, "Case Study: Generating Validation Code," where I selectively replace methods in a class to leave the developer's methods in place while replacing my generated methods.)

In a well-designed solution, you should only need to update an existing file occasionally. Typically, solutions end up having to reconcile old code with new code because the solution didn't provide a clean separation between the generated code and the developer's custom code. For this example, I keep most of the generated code in one file and provide a separate file for the developer's custom code.

For this solution, both of the files will have their names begin with "ConnectionManager." The file that holds the generated code will be named "ConnectionManager.Generation," the file holding the developer's code will be named "ConnectionManager.Customization." Initially, all I build into the solution is the ConnectionManager.Generation file.

In this solution, if the ConnectionManager.Generation file already exists, I don't want to try adding it again and catching the error: I always want to delete any existing version of the file in order to start generating the code from a blank slate. To ensure that I'm deleting the right file, I use the full pathname to the file by concatenating together the path to the project and name of the folder I added. The code looks like this:

```
string ProjectPath;
ProjectPath = System.IO.Path.GetDirectoryName(prj.FullName);
```

```
ProjectItem prji = sln.FindProjectItem(@ProjectPath + @"\" +
          codeFolder.Name + @"\ConnectionManager.Generation.cs");
if (prji != null)
{
 prji.Delete();
}
```

Adding the Template

After ensuring that the file doesn't exist, I now add the Visual Studio template that provides the base for my generation class: a class file in C#. To get this file in the right folder, I use the reference to the folder where I'm going to keep my generated code, which I retrieved earlier. This example adds the template for a non-ASP.NET project:

```
string ItemTemplatePath = sln.GetProjectItemTemplate(
                                                "Class.zip", "CSharp");
ProjectItem pji = codeFolder.ProjectItems.AddFromTemplate
            (ItemTemplatePath,"ConnectionManager.Generation.cs");
```

To enable my add-in to support ASP.NET, I need to add a different template. This code checks the project's `Kind` property and, when the project is a website, adds the correct class. Revising the previous code to handle ASP.NET projects produces the following code:

```
string ItemTemplatePath;
if (prj.Kind == "{E24C65DC-7377-472b-9ABA-BC803B73C61A}")
{
 ItemTemplatePath = sln.GetProjectItemTemplate("Class.zip",
                                                    @"Web\CSharp");
}
else
{
 ItemTemplatePath = sln.GetProjectItemTemplate("Class.zip",
                                                    "CSharp");
}
ProjectItem pji = codeFolder.ProjectItems.AddFromTemplate
           (ItemTemplatePath,"ConnectionManager.Generation.cs");
```

It's possible, for a number of reasons, that the `AddFromTemplate` method will successfully add the class file but not return a `ProjectItem`. (For

instance, if the template is a wizard, you won't get a return value because wizards don't return `ProjectItem`s.) So, after adding the item, I check to see if the reference is null; if it is, I use `FindProjectItem` to get a reference to the class file. (This also provides a check that the class file was successfully added.)

```
if (pji == null)
{
 pji = sln.FindProjectItem(@ProjectPath + @"\" +
          codeFolder.Name + @"\ConnectionManager.Generation.cs");
}
if (pji == null)
{
  applicationObject.DTE.StatusBar.Text =
                                    "Unable to add class file.";
  return;
}
```

Customizing the Template

Because the input to any code-generation solution controls the output, it's time to consider what the input for this code-generation solution looks like. I assume that the config file for the application contains a `ConnectionStrings` element, like this:

```
<connectionStrings>
   <add name="MainDB" connectionString="..." providerName="..."/>
</connectionStrings>
```

The solution should generate a class that looks like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MyProject
{
 class ConnectionManager
 {
  string MainDB
```

```
  {
   get
   {
    return System.Configuration.ConfigurationManager.
                ConnectionStrings["MainDB"].ConnectionString;
   }
  }
 }
}
```

Unfortunately, the result of adding the template for a new class in a non-ASP.NET project looks like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MyConsoleProject.Generated_Code
{
     class ConnectionManager
     {
     }
}
```

In a projectless web application, the class looks like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

/// <summary>
/// Summary description for ConnectionManager
/// </summary>
public class ConnectionManager
{
       public ConnectionManager()
       {
              //
              // TODO: Add constructor logic here
              //
       }
}
```

A number of differences exist between the template file and the class file for which I'm aiming. To get the class file I want, I must do the following:

- **Simplify the namespace.** For many projects, I will have added the class to a subfolder named Generated Code. By default, in a C# project, the folder name will be included in the class's namespace (e.g., MyProject.Generated_Code). I'd prefer not to force developers to have to drill down through the Generated_Code namespace; instead, I will have the ConnectionManager be in the project's root namespace.
- **Make the class static/shared.** Making this change allows the developer to call properties on the class without having to instantiate it.
- **Delete the constructor.** Static/shared classes are not allowed to have constructors.
- **Make the class a partial class.** Because this is created as a partial class, developers can customize ConnectionManager's behavior by adding code to a separate file.

In addition, I want to ensure that the project has a reference to the `System.Configuration` DLL. Web projects will have this reference by default but other types of project won't.

Had I used a custom template (as described in Chapter 8, "Other Tools: Templates, Attributes, and Custom Tools," and demonstrated in the case study in Chapter 10), I could omit much of the following code. However, using custom templates does make your code-generation solution dependent on having the right template installed on the developer's computer. Although the following solution requires more code, it does mean that my solution is more self-contained.

Fixing the Namespace

To simplify the `Namespace`, I first retrieve the `FileCodeModel` for the class file. If the project is a "projectless" website, I must cast the `ProjectItem` as a `VSWebProjectItem` and call its `Load` method before I can access its `FileCodeModel`. For other project types, I can just access the `FileCodeModel`; therefore, once again, the code checks to see if this is an ASP.NET project and does the right thing:

```
FileCodeModel fcm;
if (prj.Kind == "{E24C65DC-7377-472b-9ABA-BC803B73C61A}")
{
 VsWebSite.VSWebProjectItem tmpWPI;
 tmpWPI = (VsWebSite.VSWebProjectItem) pji.Object;
 tmpWPI.Load();
 fcm = tmpWPI.ProjectItem.FileCodeModel;
}
else
{
 fcm = ConnMgr.FileCodeModel;
}
```

Once the `FileCodeModel` is retrieved, I iterate through the top-level items until I find the `Namespace`. Once I find the `Namespace`, I set it to the project's `DefaultNamespace`, which I retrieved from the `Project`'s `Properties` collection. For Visual Basic projects, a `Namespace` typically isn't included in the file, but that's not a problem—if the `Namespace` isn't found, the code does nothing:

```
CodeElement2 codeClass;
foreach (CodeElement2 ce in fcm.CodeElements)
{
 if (ce.Kind == vsCMElement.vsCMElementNamespace)
 {
  ce.Name = prj.Properties.Item(
                  "DefaultNamespace").Value.ToString();
```

Because my code resets the `Namespace`'s name, there's a very real possibility that my reference to the `Namespace` may be corrupted after the change. So, after changing the `Namespace`'s name, I use this code to reacquire the reference to the `Namespace`:

```
CodeElement2 ceNamespace = (CodeElement2) fcm.CodeElements.Item
      (prj.Properties.Item("DefaultNamespace").Value.ToString());
```

Modifying the Class

To modify the class, I now find the `Class` by iterating through the `CodeElements` collection within the `Namespace` I just changed and store the reference in a variable named `codeClass`:

```
foreach (CodeElement2 ceClass in ceNamespace.Children)
{
  {
   if (ce.Kind == vsCMElement.vsCMElementClass)
   {
    codeClass = ce;
   }
  }
```

If there is no `Namespace` in the `Class` (the typical scenario for a Visual Basic file), the code acquires the reference to the `Class` and puts it in `codeClass` inside the loop that looks for the `Namespace`:

```
if (ce.Kind == vsCMElement.vsCMElementClass)
{
 codeClass = ce;
}
```

With the `Namespace` corrected (if present) and a reference to the class held in the `codeClass` variable, I now look for the class's constructor inside `codeClass`'s `Children` collection and delete it. Because I've added a C# file, I can identify the constructor by looking for a function with the same name as the class ("ConnectionManager"). For a Visual Basic application, I'd be looking for a method named `New`:

```
foreach (CodeElement2 ce in codeClass.Children)
 {
  if (ce.Kind == vsCMElement.vsCMElementFunction &&
      ce.Name == "ConnectionManager")
  {
   fcm.Remove(ce);
  }
}
```

I also need to modify the class's definition to make the class partial and shared/static. A `CodeClass2` object has the necessary functionality to make those changes. Because the code has already retrieved a reference to the class as a `CodeElement`, all that I have to do is to cast my `codeClass` reference to a `CodeClass2` object to get the functionality I need:

```
CodeClass2 cc = (CodeClass2) codeClass;
```

Now that I have a reference to a `CodeClass2` object, I make the class a partial class by setting its `ClassKind` property and a static/shared class by setting its `IsShared` property:

```
cc.ClassKind = EnvDTE80.vsCMClassKind.vsCMClassKindPartialClass;
cc.IsShared = true;
```

Adding a Reference

In order to access the `ConnectionStrings` element in the application's configuration file, non-web projects will need a reference to the `System.Configuration` assembly (website projects already have the necessary reference). To add this reference, the first step is to cast the reference to the `Project` object to a `VSLangProj.VSProject` type. Once the project is cast, a reference to the `System.Configuration` assembly can be added by name using the `References` collection's `Add` method (if the reference is already present, no error is raised):

```
VSLangProj.VSProject vsPrj;
vsPrj = (VSLangProj.VSProject) prj.Object;
vsPrj.References.Add("System.Configuration");
```

Generating Code

With the template fully customized, I can start generating the code for the properties I want to add to the class. For now, I'm going to assume that I've retrieved a single connection string name from the application's configuration file and put the connection string's name in the variable `PropertyName`. I'm also going to assume that the single line of code that the property requires is in the variable `PropertyReturnCode`. In the next section, "Reading Input," I look both at retrieving the information from the configuration file and handling multiple connection strings. The code for this example is sufficiently simple that using the CodeDom to generate the code is overkill. In the next chapter, I look at a case study where the code is sufficiently complex to justify the CodeDom.

The following code adds a property using whatever name is in the variable `PropertyName` (I omit the name for the property's setter in order to create a read-only property):

```
CodeProperty cp;

cp = cc.AddProperty(PropertyName, null,
                        vsCMTypeRef.vsCMTypeRefString, -1,
                        vsCMAccess.vsCMAccessPublic,null);
```

The design for the ConnectionManager requires the method to be static/shared. In theory, to make that change all I need to do is set the `IsShared` property on the `CodeProperty2` object that represents my newly added property. Unfortunately, in some versions of Visual Studio, the `AddProperty` method returns a `CodeProperty` object that doesn't support the `IsShared` method and can't be cast to a `CodeProperty2` object.

The solution is to use the `CodeProperty` object's `Getter` property to retrieve the `CodeFunction` object for the new property's getter, and because `CodeFunction`s do have an `IsShared` property, I can use that to make the property static/shared:

```
cp2.Getter.IsShared = true;
```

Now that the property has been added, I insert the code for the property using the `CodeEditor` object. The first step is to retrieve the `StartPoint` for the body of the property's `Getter` and, from it, create an `EditPoint`. Once the `EditPoint` is created, my next step is to delete any default code inserted into the property by the `AddProperty` method (in C#, for instance, the `AddProperty` method inserts a line of code that throws an exception):

```
EditPoint epGetter = cp.Getter.GetStartPoint(
                    vsCMPart.vsCMPartBody).CreateEditPoint();
epGetter.Delete(cp.Getter.GetEndPoint(vsCMPart.vsCMPartBody));
```

After clearing any default code, the final step is to insert any new code:

```
epGetter.Insert(PropertyReturnCode);
```

Reading Input

So far, I've just assumed that I've retrieved the inputs to the process: the names of the connections string in the app.config or web.config file. In this section, I look at retrieving that input and integrating it into the solution.

I've kept the code for this case study purposely simple to concentrate on the structure of a code-generation solution. For a case study that generates more complex code, see Chapter 10.

Processing the Configuration File

The connection strings are kept in the configuration file for the application, so my first step is to retrieve either the web.config file (for ASP.NET projects) or app.config file (for all other project types). Rather than check the project type, I use the `ProjectItems` collection to try and retrieve the app.config file; if I don't find it, I try to retrieve the web.config file. If neither is found, there are no connection strings to generate so I display a status message and exit.

Because failing to find an item in the `ProjectItems` collection raises an error, I use a `try...catch` block to determine if the configuration files are found:

```
ProjectItem cfg;


try
{
 cfg = prj.ProjectItems.Item("web.config");
}
catch
{
 try
 {
   cfg = prj.ProjectItems.Item("app.config");
 }
 catch
 {
  if (prj == null)
  {
   applicationObject.DTE.StatusBar.Text =
                                    "No configuration file.";
   return;
  }
 }
}
```

Once I've found the configuration file, the next step is to read it. The `Properties` collection for a `Project` item includes the `FullPath` to the item. Using that value, I can load the configuration file into an `XMLDocument`, as this code does:

```
System.Xml.XmlDocument dom;
dom = new System.Xml.XmlDocument();

dom.Load(@cfg.Properties.Item("FullPath").Value.ToString());
```

Adding Property Code

With the document loaded, the next step is to loop through the children of the `connectionStrings` element. For each child element, I retrieve the name attribute from the element and use that to create the property:

```
System.Xml.XmlNode ndCons =
dom.SelectSingleNode("//connectionStrings");
foreach (System.Xml.XmlNode ndCon in ndCons)
{
 string PropertyName = ndCon.Attributes["name"].Value;

 CodeProperty cp;
 cp = cc.AddProperty(PropertyName, null,
                     vsCMTypeRef.vsCMTypeRefString,
                     -1, vsCMAccess.vsCMAccessPublic, null);
 cp.Getter.IsShared = true;

 EditPoint epGetter = cp.Getter.GetStartPoint(
                      vsCMPart.vsCMPartBody).CreateEditPoint();
 epGetter.Delete(cp.Getter.GetEndPoint(vsCMPart.vsCMPartBody));
 epGetter.Insert("return " +
           "System.Web.Configuration.WebConfigurationManager." +
           "ConnectionStrings[\""+ PropertyName +
           "\"].ConnectionString;");
}
```

For a non-ASP application, the last line of code looks like this:

```
epGetter.Insert("return ConfigurationManager.ConnectionStrings[
                  \"" + PropertyName + "\"].ConnectionString;");
```

Notifying the Developer

So far, in notifying the developer, I've simply written a message to the status bar. However, that's only really appropriate for messages that provide information about ongoing processing. Where the add-in is unable to continue processing, it's more appropriate to write the message to the `TaskList`, where it will appear in the Add-In and Macros category.

Defining the Output Utility

To handle output, I use a single method that, when passed a message and a severity level, either updates the status bar or adds an item to the task list, depending on the severity level. Updating the status bar not only lets the developer using your utility know what's going on, it's also helpful in debugging—if your add-in abends, the status bar will display the last message sent to it, giving you a clue as to where in your add-in you stopped processing. Because I use this method in a variety of code-generation projects, I put it in its own class library project called CodeGenerationUtilities (this project needs references to both `EnvDTE` and `EnvDTE80`).

My utility also includes an enumeration, which I call GenerationLevel. I use it to specify the error level of the message. As a minimum, you need to support two severity levels: one for messages to be written to the status bar and one for messages to be written to the Task List. The two levels that I use are called "information" and "severe":

```
namespace CodeGenerationUtilities
{
 public enum GenerationLevel
 {
  information,
  severe
 }
```

In order to update the status bar and the Task List, my utility needs to access the `DTE2` object used by the add-in. I pass that reference to the utility in its constructor.

Handling the Task List

In the utility's constructor, I delete all related messages that may be in the Task List from previous code generations. In order to avoid deleting messages created by other code-generation utilities, I use the `TaskList`'s `SubCategory`: When adding messages I set the `SubCategory` to a value unique to the particular code-generation solution. (All my code-generation solutions set the `TaskList`'s `Category` to "Code Generation.") As a result, I can use the `SubCategory` to delete messages from previous executions of this code-generation solution. I pass the `SubCategory` to be used when adding or deleting messages into the utility's constructor and store it in a field. As a result, the constructor for the utility looks like this:

```
string subcategory;

public Utilities(DTE2 ApplicationObject, string SubCategory)
{
 applicationObject = ApplicationObject;
 subCategory = SubCategory;
 TaskList tl = applicationObject.ToolWindows.TaskList;
 foreach (TaskItem ti in tl.TaskItems)
 {
  if (ti.SubCategory == SubCategory)
  {
   ti.Delete();
  }
 }
}
```

My utility includes a `WriteOutput` method that accepts the message text to display and a `GenerationLevel` flag. If `GenerationLevel` is set to `severe`, the message is added to the `TaskList`; if `GenerationLevel` is set to `information`, the message is used to update the status bar:

```
public class Utilities
 {
  public void WriteOutput(string Message, GenerationLevel Level)
  {
   if (Level == GenerationLevel.severe)
   {
    TaskList tl = applicationObject.ToolWindows.TaskList;
    TaskItems2 tis = (TaskItems2)tl.TaskItems;
```

```
    TaskItem ti = tis.Add2("Code Generation",
                           subCategory, Message,
                           (int)vsTaskPriority.vsTaskPriorityHigh,
                           (int)vsTaskIcon.vsTaskIconCompile,
                           true,"",0,true,true,false);
   }
   else if (Level == GenerationLevel.information)
   {
    applicationObject.DTE.StatusBar.Text = Message;
   }
}
```

Using the Output Method

With the `CodeGenerationUtilities` object created, I can add a reference to the add-in so that it can use the class. To simplify code, the add-in will need a `using` statement (or an `Imports` statement in Visual Basic) that points to the new project:

```
using CodeGenerationUtilities;
```

My code-generation solution also needs a class-level variable that can hold a reference to the utility:

```
Utilities util;
```

In my add-in's constructor, I create a reference to the `CodeGenerationUtilities` object:

```
util = new Utilities(applicationObject,
                                  "ConnectionStringGenerator");
```

With that work done, I can use the `WriteOutput` method to send messages to the developer running the code-generation solution. As an example, the following call adds a message to the Task List:

```
util.WriteOutput("Unable to create Connection Manager",
                      GenerationLevel.severe);
```

Supporting Customization

As I noted at the start of this case study, part of this solution includes giving the developer the ability to modify the connection string retrieved from the configuration file. There are at least two ways to provide this option:

- Add a second partial class (the "customization" class) where the developer can add code to modify the connection string. The class holding the generated code calls methods in this second class before returning the connection string to the calling application. The developer can add code inside these methods to modify the connection string.
- Allow the developer to inherit from our generated class. Again, our generated code would call methods that allow the developer to modify the connection string before the string is returned. However, with this design, the developer would override those methods to add his or her own code.

For this case study, I use the first strategy. As part of that strategy, I add a second file to the project (named ConnectionManager.Customization) where developers can put their custom code.

I also allow the developer to turn customization on and off so that when the developer doesn't need to modify the connection string through Visual Studio's Options dialog, the customization support (e.g., the ConnectionManager.Customization file) won't be generated.

Customizable Code

When customization is turned on, the property generated calls a method and passes it the connection string. The property then returns whatever is passed back by the method. A typical example of the generated code with customization support looks like this:

```
public static string Northwind
{
 get
 {
  return NorthwindCustomization(
         System.Web.Configuration.WebConfigurationManager.
             ConnectionStrings["Northwind"].ConnectionString);
 }
}
```

The corresponding customization class contains stubs for the customization methods:

```
public static partial class ConnectionManager
{
 public static string NorthwindCustomization(string ConnectionString)
 {
  return ConnectionString;
 }
}
```

Developers can now put any code to modify the connection string in these stubs. To ensure that the developer never loses any code, my code never deletes the customization class. If a customization stub doesn't exist, the code-generation process will add the stub. However, no compile error is raised if a developer renames (or deletes) a connection string that he or she has written customized code for. Unfortunately, the customized code will never be called.

The add-in offers one other customization option. Although the add-in's default implementation is a static/shared class, developers may find that too restrictive when they start adding their custom code. In order to give the developers more options, I also allow them to turn off the static/shared option.

Accepting Input

Rather than expect the developer to specify these options for each generation, I let the developer set the customization options in the Tools | Options dialog. For a complete solution, the options should be stored on a project-by-project basis so the dialog for these choices should be a list of projects showing the choice for each option. However, that would take the focus of this case study into the realm of Windows Form programming and away from creating an effective code-generation implementation, so this example just supports a global setting that applies to all projects.

Defining the Options Dialog

My first step in adding to the Tools | Options dialog is to create a separate project (named ConnectManagerUI) to hold the user control that becomes part of the Tools | Options dialog. Because, even for testing purposes, this project's DLL must go into the Add-Ins library, I create a new class library

project and set the output path on the Tools | Options | Build dialog to ...\Visual Studio *version*\Addins\.

In order to have the user control loaded by Visual Studio, I add the following elements to my add-in project's .Addin files (this code assumes that the user control will be called ConnectionManagerOptions). The Tools | Options dialog uses the values in the `Category` and `SubCategory` elements to create the TreeView on the left side of the dialog that lets the developer navigate to my user control. I also use the `Category`/`SubCategory` values in my add-in's code to retrieve the options the developer sets:

```
<ToolsOptionsPage>
 <Category Name="Code Generation">
  <SubCategory Name="Connection Manager">
   <Assembly>ConnectionManagerUI.dll</Assembly>
    <FullClassName>ConnectionManagerUI.ConnectionManagerOptions
    </FullClassName>
  </SubCategory>
 </Category>
</ToolsOptionsPage>
```

Saving Developer Choices

It's my responsibility to save and retrieve the choices entered by the developer in the Tools | Options dialog. To support that, I add a class to my CodeGenerationUtilities project with methods that save and retrieve string values to and from the Windows registry. That class looks like this:

```
namespace CodeGenerationUtilities
{
 public class Utilities
 {
  public static string GetValue(string Name)
  {
   Microsoft.Win32.RegistryKey key;

   key = Microsoft.Win32.Registry.CurrentUser.OpenSubKey(
                 @"SOFTWARE\Microsoft\VisualStudio\9.0", false);
   return key.GetValue(Name, "").ToString();
  }

  public static void SaveValue(string Name, string value)
 {
```

```
  Microsoft.Win32.RegistryKey key;

  key = Microsoft.Win32.Registry.CurrentUser.OpenSubKey(
                    @"SOFTWARE\Microsoft\VisualStudio\9.0", true);
  key.SetValue(Name, value,
                       Microsoft.Win32.RegistryValueKind.String);
 }
}
```

Option Manager Class

Before creating the user control, I also create a class in the same project as the user control to manage the values entered by the developer. In addition to simplifying the code in the user control, this option manager class is required if I'm going to pass the values saved by the user control to the add-in that generates the code.

The option manager class has one property for each value I allow the developer to set in the user control and uses the `SaveValue` and `GetValue` methods in my `Utilities` class to save data in the Windows registry as strings. The code in the option manager class sets the names that these values will be saved under in the Windows registry. The naming convention that I use is the word "Generate," followed by the name of the code-generation solution, followed by the property name.

This option manager class for this case study has properties for turning customization support on or off (`SupportCustomization`, which saves its value under the name `GenerateConnectionManagerSupportCustomization`) and specifying whether the class and property should be static/shared (`IsStatic`, which saves its value under the name `GenerateConnectionManagerIsStatic`):

```
namespace ConnectionManagerUI
{
 public class ConnectionStringProperties
 {
  public string SupportCustomization
  {
   get
   {
    return Utilities.GetValue(
         "GenerateConnectionManagerSupportCustomization");
   }
   set
   {
```

```
    Utilities.SaveValue(
        "GenerateConnectionManagerSupportCustomization", value);
   }
  }

  public string IsStatic
  {
   get
   {
    return Utilities.GetValue(
        "GenerateConnectionManagerIsStatic");
   }
   set
   {
    Utilities.SaveValue(
        "GenerateConnectionManagerIsStatic", value);
   }
  }
 }
}
```

Creating the User Control

I'm finally ready to add the user control that will appear in the Tools | Options dialog (see Figure 9-2). The user control has two check boxes, one for each of the two options offered by this add-in: whether a customization file will be generated and whether the generated classes should be static/shared.

I must add two attributes to the user control to have it work well with the Tools | Options (`ComVisible` and `ClassInterface`). In addition, the user control must implement the `EnvDTE.IDTToolsOptionsPage` interface. This code shows the resulting definition for the user control for this case study:

```
namespace ConnectionManagerUI
{
 [System.Runtime.InteropServices.ComVisible(true)]
 [System.Runtime.InteropServices.ClassInterface(
    System.Runtime.InteropServices.ClassInterfaceType.AutoDual)]
public partial class ConnectionManagerOptions: UserControl,
                                                EnvDTE.IDTToolsOptionsPage
 {
```

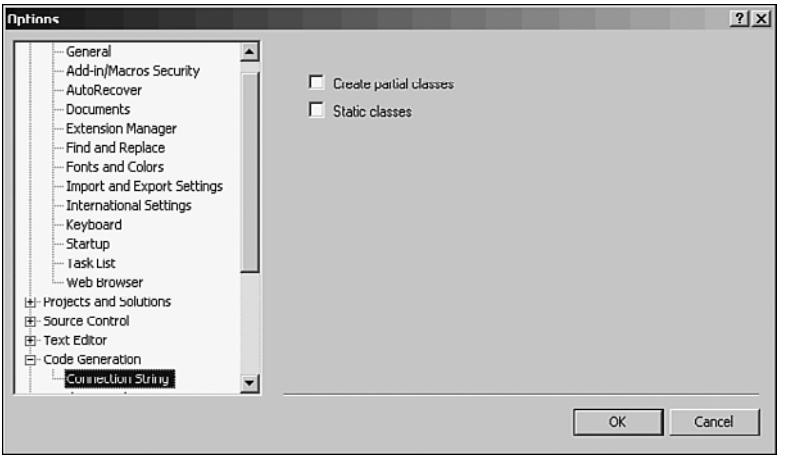

FIGURE 9-2 The user control for the case study allows the developer to turn support for customization on or off and to specify whether the generated class is static/shared.

In the user control, I take advantage of the option manager class that I created earlier to do most of the user control's work. I instantiate that class in my user control's constructor:

```
public ConnectionManagerOptions()
{
 InitializeComponent();
 opts = new ConnectionStringProperties();
}
```

Implementing the User Control Interface

The `IDTToolsOptionsPage` interface adds several methods to the user control, but I only need to put code in four of them. I add code to the `OnAfterCreated` event to retrieve the current values for the property and to the `OnOk` event to save the current values. In these events, I just call the appropriate methods on my option manager class (with a little extra code to initialize the page when the user control is called for the first time):

```
public void OnAfterCreated(EnvDTE.DTE DTEObject)
{
 if (opts.IsStatic == "true"|| opts.IsStatic == "")
 {
```

```
  this.StaticCheckbox.Checked = true;
 }
 else
 {
  this.StaticCheckbox.Checked = false;
 }

 if (opts.SupportCustomization == "true")
 {
 this.CustomizationCheckbox.Checked = true;
 }
 else
 {
  this.CustomizationCheckbox.Checked = false;
 }
}

public void OnOK()
{
 if (this.StaticCheckbox.Checked)
 {
  opts.IsStatic = "true";
 }
 else
 {
  opts.IsStatic = "false";
 }

 if ( this.CustomizationCheckbox.Checked)
 {
  opts.SupportCustomization = "true";
 }
 else
 {
  opts.SupportCustomization = "false";
 }
}
```

Because I intend to pass the values collected in the user control to an add-in running in Visual Studio, I also implement the interface's `GetProperties` method. All that I have to do is to set the `PropertiesObject` passed to this routine to an instance of my option manager class:

```
public void GetProperties(ref object PropertiesObject)
{
 PropertiesObject = opts;
}
```

Integrating with the Add-In

With the work on the user control complete, the developer can choose his or her options in the Tools | Options page. I access the developer's choices by retrieving a `Properties` object from the `applicationObject`, specifying the `Category` and `SubCategory` I set in the .add-in file's `ToolsOptionsPage` element. I typically end up using these options throughout my add-in, so I usually declare the `Properties` object at the class level:

```
Properties props;
```

I then retrieve the options in the add-in's constructor. To retrieve the options set through the user control, I pass the `Category` and `SubCategory` I set in the `ProvideOptionPage` attribute on the user control to the `get_Properties` method on the `applicationObject`. (In Visual Basic, you read the `Properties` property.) In the `get_Properties` method, the `SubCategory` value is passed to a parameter called `PageName`. For this case study, the `Category` is "Code Generation" and the `SubCategory` is "Connection Manager":

```
props = applicationObject.get_Properties["Code Generation",
                                         "Connection Manager"];
```

To retrieve any particular property, I pass the property name from my data manager object to the `Properties` object's `Item` method. This example, for instance, retrieves my `IsStatic` method from my option manager class and, because the value returned by the property is a string, converts it into a Boolean value:

```
bool IsStatic;

if (props.Item("IsStatic").Value.ToString() == "true")
{
 IsStatic = true;
}
else
{
 IsStatic = false;
}
```

The resulting values can be used to control code generation. For instance, this example uses the `IsStatic` value to control whether the class is declared as static/shared:

```
if (!IsStatic)
{
 cc.IsShared = true;
}
```

Generating Custom Code

Working with a file that holds code written by the developer requires a different strategy than a file holding only code you generate. In general, it's *never* okay to delete a developer's code, but it is okay to make the code invalid or irrelevant.

As an example, in this case study the developer may add custom code to work with the Northwind property that is tied to the Northwind connection string. If the developer then deletes the connection string named "Northwind" from the configuration file and ConnectionManager is regenerated, my solution will re-create the ConnectionManager.Generation file without the Northwind property.

Without the Northwind property in place, the developer's custom code is orphaned and will never be called—but that's not a problem (at least, it's not *your* problem). Even if removing the generated Northwind property prevents the solution from compiling because of problems with the custom code (not the case with this solution), the problem is—from the developer's point of view—solvable: When the compile fails, the developer will get a message pointing to the offending custom code. The developer can then modify or delete the code.

What would not be a good idea would be to "helpfully" delete the developer's custom code. After all, the developer may intend to move his or her orphaned custom code to another custom routine—if my solution deletes the code, that option is no longer available to the developer.

In the customization file, the general strategy is to first check before adding any custom code to see if it's already present. If the code is present, the solution should leave the code alone; if the custom code isn't present, the solution should generate whatever support code is part of the code-generation solution. If the developer wants to have any support for custom code regenerated, all the developer has to do is delete the relevant custom code. With the custom code gone, the solution will regenerate any necessary support code.

Adding Custom Code

In the case study, the first place where I implement this strategy is in adding the customization file. For the file holding the generated code, the file is always deleted and re-created. For the customization file, on the other hand, if the file is present, the solution just retrieves a reference to it; only if the customization file isn't already present does my solution generate the customization file. This code checks to see if customization is being supported and, if it is, implements that strategy:

```
if (IsCustomized)
{
 pjic = sln.FindProjectItem(@ProjectPath + @"\" +
       codeFolder.Name + @"\ConnectionManager.Customization.cs");
 if (pjic == null)
 {
   if (prj.Kind == "{E24C65DC-7377-472b-9ABA-BC803B73C61A}")
   {
    ItemTemplatePath = sln.GetProjectItemTemplate("Class.zip",
                                                  @"Web\CSharp");
   }
   else
   {
    ItemTemplatePath = sln.GetProjectItemTemplate("Class.zip",
                                                   "CSharp");
   }
   pjic = codeFolder.ProjectItems.AddFromTemplate(
         ItemTemplatePath, "ConnectionManager.Customization.cs");
 }
}
```

The same process is followed when adding the support stubs inside the customization file: Stubs are only added if they're not already present.

Tying Generation to Events

Rather than use a menu item to trigger code generation, a better solution for this case study is to tie the code generation to events in Visual Studio. The obvious choice for this case study is to check for changes in the configuration file: Whenever the configuration file is closed, for instance, the code could check for the presence of connection strings and regenerate the ConnectionManager. This is the strategy used for generating the code

behind the .NET DataSet designer: The code is generated when the designer is closed (and, in Visual Studio 2008 and 2010, when the focus shifts away from the DataSet's visual designer).

However, the developer also needs a way to force the ConnectionManager to be regenerated if only for those situations where the developer wants to make a change and leave the configuration file open. You could leave the menu item in place (or add a button to the ConnectionManager's Tools | Options dialog). However, a better solution is to tie the code generation into the build process.

Integrating with Builds

Integrating with the build process is the easier of the two events to set up, so I look at that option first. The first step is to declare a class-level variable to hold a reference to the events package that references build events. For a build-related event, that variable is declared with the `BuildEvents` data type, as this code does:

```
EnvDTE.BuildEvents BuildE;
```

In order to tie the code-generation property into Visual Studio events, I have to wire up the events when the add-in is loaded by Visual Studio. I have a couple of choices here: I can either use the add-in's constructor (called from the `Connect` method in C# or the `New` method in Visual Basic) or the `OnConnection` method. I use the `OnConnection` method because I can check the `connectMode` parameter passed to the method to ensure that the method is being called in setup mode. Just to be safe, though, I also check that I haven't already set up the event by seeing if my class-level variable is set to null.

This code retrieves the `BuildEvents` package and then ties a method in the connect class (which I've named `BuildE_OnBuildBegin`) to the `OBuildBegin` event:

```
if (connectMode == ext_ConnectMode.ext_cm_UISetup)
{
 if (BuildE == null)
 {
  BuildE = _applicationObject.Events.BuildEvents;
  BuildE.OnBuildBegin += new
        _dispBuildEvents_OnBuildBeginEventHandler(BuildE_OnBuildBegin);
 }
```

In my `OnBuildBegin` method, I need to create my generation class (`DatabaseUtilities`) and call my code-generation method

(`GenerateConnectionManager`) whenever the project is being rebuilt. To determine whether the project is being rebuilt, I can check the two flags passed into the event handler:

- **`BuildScope`**—Reports the scope of the build (batch, project, or solution)
- **`BuildAction`**—Type of build (`Clean`, `Build`, `Rebuildall`, `Deploy`)

If I created temporary files or folders that I didn't automatically delete as part of the code-generation process, I should remove those when the event is called with `BuildAction` set to `Clean`. However, for this solution, the `Clean` action is the one `BuildAction` where I don't want to regenerate ConnectionManager:

```
void BuildE_OnBuildBegin(vsBuildScope BuildScope,
                         vsBuildAction BuildAction)
{
 if (BuildAction != vsBuildAction.vsBuildActionClean)
 {
  DatabaseUtiltiies dbu;
  dbu = new DatabaseUtiltiies(_applicationObject,
                              _addInInstance);
  dbu.GenerateConnectionManager();
 }
}
```

Integrating with Documents

To catch the events that fire when the configuration file is opened or closed, I first have to catch the events fired when any document is opened or closed. I add another field (named `docMaster`) to my class to hold the `DocumentEvents` package for this "master" document event routine. Eventually, I'm going to need a reference for the event that ties to my configuration file, so I also add a field (named `docConfig`) to hold that reference:

```
EnvDTE.DocumentEvents docMaster;
EnvDTE.DocumentEvents docConfig;
```

In the `OnConnection` method, when the method is called in setup mode, I check to see if I've set the event package; if I haven't, I set the reference. Once I've set the reference, I attach a method (which I've called `docMaster_DocumentOpened`) to the `DocumentOpened` event. The `DocumentOpened` event is a filtered event: Because I pass a reference to a document to this

event, the event will only fire when that specific document is opened. For my "master" event handler, however, I want to catch events for all documents, so I pass a null as part of wiring up the event:

```
if (docMaster == null)
{
 docMaster = (EnvDTE.DocumentEvents)
             _applicationObject.Events.get_DocumentEvents(null);
 docMaster.DocumentOpened +=new
                 _dispDocumentEvents_DocumentOpenedEventHandler(
                                             docMaster_DocumentOpened);
}
```

In my `docMaster_DocumentOpened` event handler, I want to wire up an event routine that will fire when the configuration file is closed. (This may be just the initial version of this handler: If I expand this solution, I can add more code in this handler to check for other documents that I'm interested in and wire up events for them also.) I first check to see if I've already set an event for the configuration file by checking the field where I hold the reference (`docConfig`). If that field is null, I then check the `Document` parameter passed to the event handler to see if the document being opened is either the web.config or app.config file:

```
void docMaster_DocumentOpened(Document Document)
{
 if (docConfig == null && Document.Name == "app.config")
 {
```

If the document being opened is either of the configuration files, I get a reference to the `DocumentEvents` package as I did in setting up the master event handler. This time, however, I filter the event by passing the `Document` object that represents the configuration file. In this case study, I want to capture the `DocumentClosing` event so I wire up the `DocumentClosing EventHandler` to a routine I've named `docConfig_DocumentClosing`:

```
  docConfig = (EnvDTE.DocumentEvents)
          _applicationObject.Events.get_DocumentEvents(Document);
  docConfig.DocumentClosing +=new
              _dispDocumentEvents_DocumentClosingEventHandler(
                                        docConfig_DocumentClosing);
 }
}
```

In my `docConfig_Closing` event handler, as before, I create my code-generation class and call the method that creates the connection manager. In addition, at the end of the routine, I set the reference for this handler to null:

```
void docConfig_DocumentClosing(Document Doc)
{
 DatabaseUtilities dbu;
 dbu = new DatabaseUtiltiies(_applicationObject, _addInInstance);
 dbu.GenerateConnectionManager();
 docConfig = null;
}
```

Generating a Simple Class

This chapter has demonstrated a complete—though simple—code-generation solution. Principally, this solution hasn't dealt with having multiple config files open at the same time, and I've deliberately restricted the objects I've used to keep the toolkit required for understanding this chapter small. I've also assumed that the generated code will always be in C#. (Although, because the solution generates so little code, extending it to handle Visual Basic—or any other language—would be very simple.) I also haven't spent much time on structuring code—the focus of the project is to concentrate on the code-generation process. The case study in the next chapter goes beyond this solution to handle multiple documents, using a larger toolkit, generating more complex code, and supporting both C# and Visual Basic through the CodeDom.

You can download the code for this case study from my website (www.phvis.com) and www.informit.com.

INDEX