Microsoft®

Working with Microsoft® Visual Studio® 2005

Craig Skibo
Marc Young
Brian Johnson

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2006 by Microsoft Corporation

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain named, e-mail address, logo, person, place, or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without express written permission of Microsoft Corporation.

Microsoft, ActiveX, Authenticode, IntelliSense, Microsoft Press, MSDN, Outlook, Visual Basic, Visual C#, Visual C++, Visual J#, Visual Studio, Visual Web Developer, Win32, Windows, and Windows Media are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

This book expresses the authors' views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

To my mom and dad, Pepper and Kathy Johnson.
Thanks for everything.
–B.J.

To my parents, Al and Jan, my brother, Brian, and all the friends
who have supported me over the years.
–C.S.

To Julia, Max, and Brigitte–kisses, hugs, and ladybugs.
–M.Y.

Contents at a Glance

Table of Contents

Acknowledgments

It might not take a village to write a book, but it takes a good sized team to put it together, so first of all we want to thank our editor, Devon Musgrave, for all the hard work that he put into this book. It's amazing how much work he can get out people purely though force of will. In addition, we thank Joel Rosenthal, who copy edited, and our good friend Bill Teel, who worked on the graphics. Thanks also to Ben Ryan and Elden Nelson for acquiring the book and to Carl Diltz and Elizabeth Hansford for production support. We also give a huge thanks to our internal reviewers: Prasadi de Silva, Mark Kenworthy, Nishan Jebanasam, Jeremy Jones, Sean Laberee, Tarek Madkour, Chad Royal, and Phil Taylor. Finally, thanks to Prashant Sridharan, Marie Hagman, Doug Hodges, the Visual Studio Extensibility Team, the Visual Studio Content Installer Team, and the Visual Studio Project team.

Introduction

The Microsoft® Visual Studio® integrated development environment (IDE) is arguably one of the most powerful and complex development tools in the world. In 2002, Craig Skibo, Marc Young, and I decided that a book was needed to show how developers could really extend this IDE and make it their own. That book was called *Inside Microsoft Visual Studio .NET 2003*. This book, *Working with Microsoft Visual Studio 2005*, is an update to that book.

Working with Microsoft Visual Studio 2005 won't teach you programming. It will however, help you to understand the IDE and the project system, and it will tell you a lot about how you can adopt the IDE to meet your specific needs.

Support information for the book is located at the end of this introduction. If you want to send feedback or suggestions to me directly, feel free to send an e-mail to *brianjo@microsoft.com*.

Who Is This Book For?

This book is for developers who want to learn a little bit more about the Visual Studio 2005 IDE and who want to extend the capabilities of that IDE with macros and add-ins.

System Requirements

You'll need the following hardware and software to build and run the code samples for this book:

- Microsoft Windows ® XP with Service Pack 2, Microsoft Windows Server ™ 2003 with Service Pack 1, or Microsoft Windows 2000 with Service Pack 4
- Microsoft Visual Studio 2005 Standard Edition
- 600 MHz Pentium or compatible processor (1 gigahertz Pentium recommended)
- 192 MB RAM (256 MB or more recommended)
- Video (800 x 600 or higher resolution) monitor with at least 256 colors (1024 x 768 High Color 16-bit recommended)
- CD-ROM or DVD-ROM drive
- Microsoft Mouse or compatible pointing device

Prerelease Software

This book was reviewed and tested against the August 2005 Community Technical Preview (CTP) of Visual Studio 2005. This book is expected to be fully compatible with the final release of Visual Studio 2005. If there are any changes or corrections to this book, they will be collected and added to a Microsoft Knowledge Base article. See the "Support for This Book" section in this Introduction for more information.

Technology Updates

As technologies related to this book are updated, links to additional information will be added to the Microsoft Press Technology Updates for Books Web page. Visit this page periodically for updates on Visual Studio 2005 and other technologies.

http://www.microsoft.com/mspress/updates/

Code Samples

All of the code samples discussed in this book can be downloaded from the book's companion content page at the following address:

http://www.microsoft.com/mspress/companion/D09-00051/

Support for This Book

Every effort has been made to ensure the accuracy of this book and the companion content. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article. To view the list of known corrections for this book, visit the following article:

http://support.microsoft.com/kb/905532

Microsoft Press provides support for books and companion content at the following Web site:

http://www.microsoft.com/learning/support/books/

Questions and Comments

If you have questions, comments, or ideas regarding the book or the companion content, or questions that are not answered by visiting the sites above, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com

Or via postal mail to

Microsoft Press
Attn: Working with Microsoft Visual Studio 2005 Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through the above addresses.

Chapter 1

Introducing Visual Studio 2005

In this chapter:

In this chapter, we'll provide a brief overview of Microsoft® Visual Studio® 2005. We'll show you some of the features of the integrated development environment (IDE) to provide some context for the extensibility and customization discussion throughout the rest of the book. And finally, we'll discuss the extensibility features that make Visual Studio 2005 an extremely attractive tool for programmers who are looking to customize and extend their development environment.

What Is Visual Studio 2005?

Visual Studio 2005 is the latest version of Microsoft's Visual Studio line of development tools. Visual Studio 2005 has a long lineage, running from its early roots as the IDE that hundreds of thousands of Microsoft Visual Basic® developers used on Microsoft Windows® starting in the early 1990s. A second ancestor, Microsoft Visual C++® 1.0, was released in 1993 and became the standard platform for C++ developers worldwide. These two IDEs were finally united in Microsoft Visual Studio .NET 2002 and Visual Studio .NET 2003. Visual Studio .NET 2002 was the first IDE for the Microsoft .NET platform and was used to develop applications for the 1.0 version of that product. Visual Studio .NET 2003 was used to build .NET Framework 1.1 applications and added a number of enhancements to the original product.

Visual Studio 2005 uses the .NET Framework 2.0 as the class library and runtime for applications built in the managed languages that ship with the IDE. These languages include Visual Basic 2005, Microsoft Visual C#® 2005, Visual C++ 2005, and Microsoft Visual J#® 2005. In the case of Visual Basic, Visual C#, and Visual J#, all applications built with Visual Studio 2005 require the .NET Framework 2.0 common language runtime (CLR), though it is possible to target an application to a previous version of the CLR. Visual C++ 2005 can be compiled as a managed .NET Framework application, requiring the CLR, or it can be compiled as a native application, meaning that no runtime is required for the application to run in Windows.

The .NET Framework and the CLR offer a number of features that developers can take advantage of when building programs. These include an extremely large and rich class library from which you can build applications, a runtime that takes care of memory management through efficient garbage collection, and a flexible security model that allows administrators to control the execution of code deployed to corporate networks and the Internet.

Developing for .NET

One purpose of the .NET Framework is to simplify application development and deployment. This extends to applications that are run locally or remotely or that are distributed over the Internet. This simplification is achieved through a CLR that provides a managed execution environment available to any language that targets the runtime. The functionality this execution environment provides is made available to these languages through the .NET Framework class library.

The Common Language Specification (CLS) specifies what a .NET-compliant language must provide to the system. The common type system (CTS) ensures that any types created by a language conforming to the CLS can be consumed by any other CLS-compliant language.

Languages that target the CLR are compiled to Microsoft intermediate language (MSIL). These applications are compiled as PE (portable executable) files and DLLs, so to users they look just like any Windows-based applications. The MSIL code in these files is then JIT-compiled to machine instructions locally at run time. All of this means that any CLS-compliant language that targets the CLR will look like any other language to the runtime and can act and be treated as a first-class citizen. For example, a Visual Basic program will have the same base functionality as a Visual C# program or even a managed C++ program.

Tip For a detailed look at the architecture of .NET, take a look at *Applied Microsoft .NET Framework Programming* by Jeffrey Richter (Microsoft Press®, 2002).

The managed CLR environment provides some other significant advantages. It's designed to help eliminate versioning conflicts. It's designed to provide an environment that ensures that code is executed safely. And finally, it's designed with an API that is targetable from both Windows-based and Web-based applications.

You don't need to think about the .NET Framework as a monolithic virtual machine that requires constant care and feeding. The .NET Framework provides an environment that can be hosted by unmanaged components. The unmanaged components (such as Microsoft Internet Explorer and the Microsoft ASP.NET runtime) load the CLR and execute the managed code. The managed CLR provides garbage-collection services and security on a number of levels.

For corporate developers, this runtime solves a huge problem. In many shops, the Visual Basic programmers, the C/C++ programmers, and the COBOL programmers are all segregated. They meet to figure out how to functionally interoperate, but in a number of ways they work as individual teams inside the same space. In a shop that targets the CLR, development becomes a little more manageable. The same .NET Framework class library is available across languages. The CTS in the class library ensures that components can be easily shared between .NET languages, as shown in Figure 1-1. These components can even be exposed as Web services.

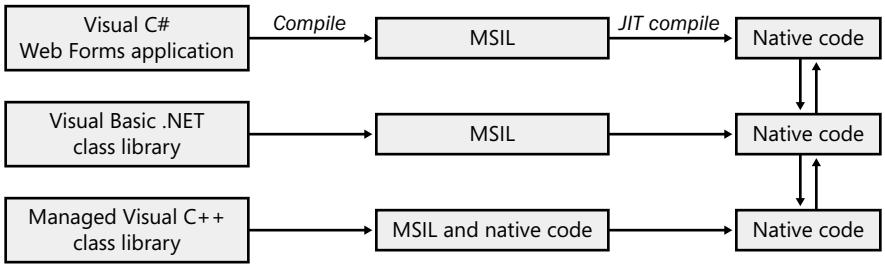


Figure 1-1 .NET allows different languages to target a managed environment and to interoperate securely and efficiently.

The CLR provides a target that's available from most of the major programming languages used today. Visual Studio 2005 allows you to build .NET applications in the languages mentioned previously. You can also add support for a number of languages that are available from third-party software vendors, including COBOL, Eiffel, Python, and Perl.

Building Native Applications

In addition to building top-of-the-line .NET Framework applications, Visual Studio 2005 ships with arguably the best C++ compiler in the world. The compiler allows developers to build high-performance native applications that can take advantage of the latest hardware features shipping today. Most native applications on Windows are built using the C++ libraries in addition to the system libraries in Windows. Native applications are often built using the Microsoft Foundation Classes (MFC) and the Abstract Template Library classes (ATL). In Visual Studio 2005, these libraries are version 8.0 and are compiled using the updated security and performance features of the Visual C++ compiler.

.NET Framework and *native* describe only the base functionality of the applications that you can build using Visual Studio 2005. In either case, you can create running applications with Notepad and the Command Window. The power of Visual Studio 2005 lies in its ability to empower users to build, test, and debug powerful applications quickly and easily.

Visual Studio 2005 Features

In this section, we'll present an overview of the Visual Studio 2005 feature set. We'll look at some of these features in more detail in the next few chapters of the book. Here we'll define some common terms that we can use to describe the different parts of the IDE. Visual Studio 2005 is a fairly large and complex product. The terms used to describe the IDE are helpful for developers who are working to understand the tool and, perhaps more important, for developers who will eventually extend the IDE through macros and add-ins.

Editors, Designers, and Tool Windows

The windows in the Visual Studio 2005 IDE fall into two major groups. *Document windows* are windows that usually appear tabbed in the center of the IDE and that contain editors, designers, Web pages, or Help topics. *Tool windows* are windows in the IDE that present utility functions to the programmer. The tool windows include Solution Explorer, the Class View window, and the Properties window. Tool windows are distinct from editors and designers in the way they dock around the sides of the IDE.

Note The extensibility API built into Visual Studio 2005 allows programmers to create tool windows for use with language packages installed into the IDE. The editors and designers in the IDE can be accessed through this API, but the extensibility model doesn't allow the creation of new document window types. For that you'll need to look into the Visual Studio software development kit (SDK), which we'll describe at the end of this chapter.

Figure 1-2 shows a typical developer setup.

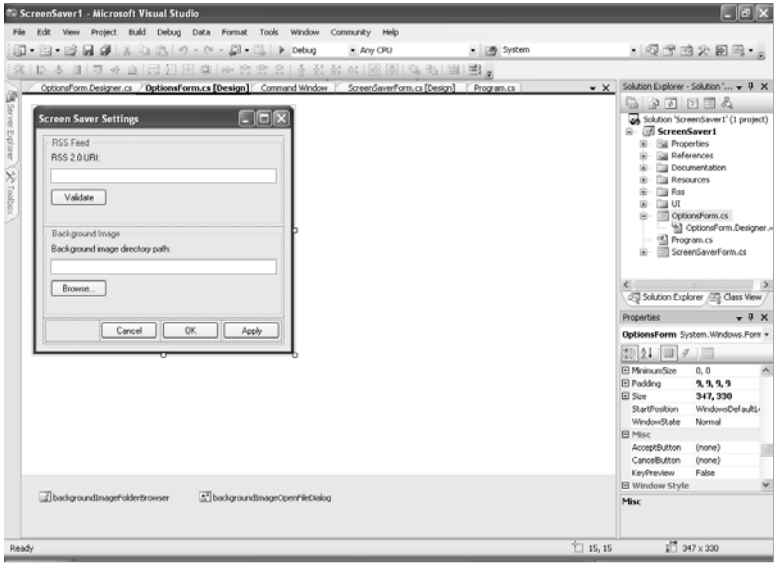


Figure 1-2 A typical solution in Visual Studio 2005

Note All of the managed languages that ship with Visual Studio 2005 feature designer support in the IDE.

The Start Page

The first time you run Visual Studio 2005, you're presented with a Start Page, which contains links to previously opened projects and information relevant to the preferred collection of settings for the developer type chosen the first time the IDE is opened. A *settings collection* is a window, keyboard, and Help layout that's tailored to a specific type of programmer. You're presented with a choice of settings when you first run Visual Studio 2005. You can change the settings that you want to run under by opening the Import and Export Settings Wizard, available from the Tools menu. This wizard allows you to back up and to transfer your settings and customization between computers. You can use this wizard to reset your settings by choosing Reset All Settings, as shown in Figure 1-3.

Figure 1-3 Resetting the IDE in the Import and Export Settings Wizard

Figure 1-4 shows the initial settings you can choose from in Visual Studio 2005.

In Chapter 6, we'll discuss customizing the IDE and saving your settings as a backup or for reuse on other computers.

The Editor

In talking with members of the team that developed the base editor in Visual Studio 2005, it's clear that they understand that programmers live in the editor. It's where the most important programming work is done. To this end, the Visual Studio team has continued to work hard to create a code editor that's one of the best available today. To a great extent,

they have succeeded in this goal, largely because of enhancements to the extensibility features of the IDE. These enhancements include an extremely powerful extensibility programming model, a macro recording facility, and a dedicated Macros IDE. The Visual Studio 2005 extensibility model is a major focus of the book because it's what we use to customize and to add functionality to the IDE.

Figure 1-4 Choose from a number of different settings collections to begin to customize the IDE.

Other features in the editor include code snippets, smart tags, change tracking, outlining, line numbering, and a really outstanding search-and-replace facility, all of which are discussed in detail in Chapter 3.

A new feature in Visual Studio 2005 allows you to tab through your open documents and tool windows by pressing Ctrl+Tab. In Figure 1-5, you can see the dialog box that appears when you press this combination. You can use the arrow keys or the Tab key to move through the listed windows as you hold down the Ctrl key.

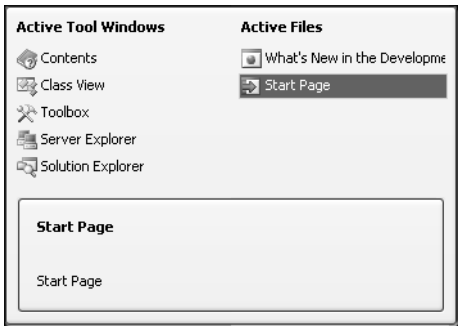


Figure 1-5 Use Ctrl+Tab to get a list of open windows in the IDE.

Designers

Visual Studio 2005 offers four major types of designers: Windows Forms designers, which let you create Windows Forms applications visually; Web Forms designers, which help you create WYSIWYG ASP.NET Web Forms applications; the Component Designer, which is used to build server-side components for enterprise solutions; and the XML Designer, which makes it easy for programmers to work with XML Schema Definition files.

In Visual Studio 2005, all languages provide designers for .NET application creation. This means that you can design your Windows Forms and Web Forms in the same language you use to write your most important algorithms. In the past, it was common for developers to create the front end of their applications by using a visual tool such as Visual Basic and to write the back end in Visual C++. Because of the way that .NET assemblies interoperate, you're still free to do your forms layout and library writing in different languages, but you're no longer forced to work that way.

Tool Windows

Tool windows are the nondocument windows in the IDE that provide you with information and utility functionality as you work. The IDE has a large number of tool windows, and you can access them easily using keyboard shortcuts, the Command Window, and menu commands. The following are the most commonly used tool windows in Visual Studio 2005. These tool windows are presented with their associated default keyboard shortcuts.

Solution Explorer (Ctrl+Alt+L) The Solution Explorer window is arguably the most important tool window in Visual Studio 2005. In Visual Studio 2005, nearly all the work done by a programmer revolves around a solution. A *solution* is a collection of *projects*, which are themselves collections of files. It's through Solution Explorer that you'll get access to the files in your projects. Here you'll add new classes and files to projects and even new projects to larger solutions. Figure 1-6 shows a project in the Solution Explorer tool window.

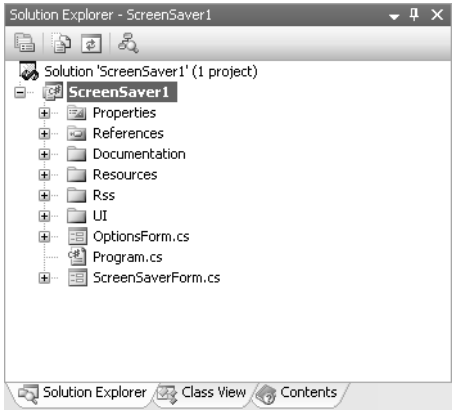


Figure 1-6 A ScreenSaver Starter Kit project in Solution Explorer

Class View (Ctrl+Shift+C) The Class View window provides you with a logical view of the classes in your solution. If you're working with larger projects, you might find it easier to navigate through your solutions by using Class View than by using Solution Explorer. Figure 1-7 shows the solution from Figure 1-6 in Class View.

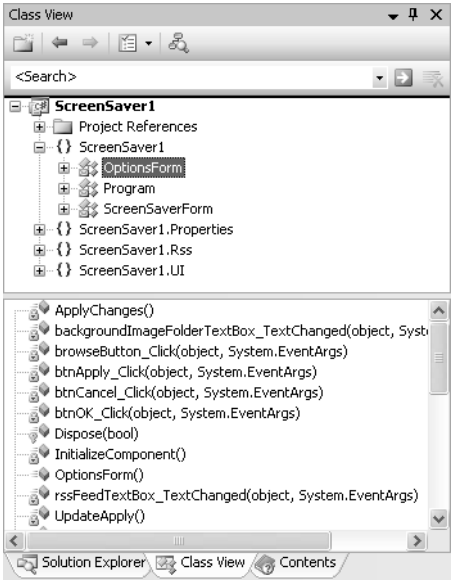


Figure 1-7 The Class View window gives you an alternative view of the objects in your solution.

Properties (F4) In the Properties window, you can get and set properties for the user interface items that you add to Windows Forms and Web Forms applications. You can also use this window to set properties for solutions, projects, and files that you have selected in Solution Explorer. Figure 1-8 shows the Properties window for a project.

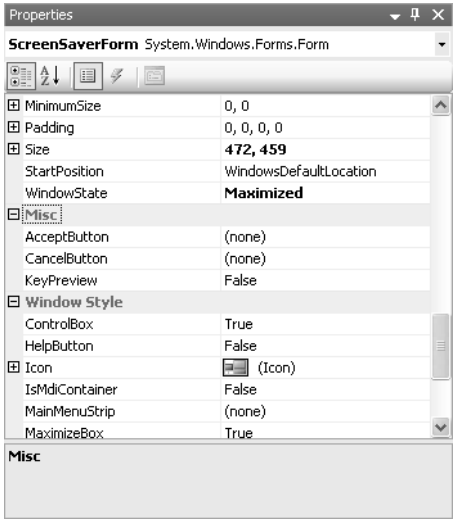


Figure 1-8 You can use the Properties window to set properties of components, projects, and solutions.

Server Explorer (Ctrl+Alt+S) You use the Server Explorer window to access data sources and information on your local machine and on remote servers. Through this window, you can make data connections, access performance counters and event logs, and even manage system services. Even when used locally, this tool can save you a great deal of time, letting you easily start and stop system services that you're testing and access system logs. In Figure 1-9, you can see two machines available in Server Explorer. The first machine is a remote test server. The second machine is the local machine on which Visual Studio 2005 is running.

Note Keep in mind that you'll need the proper level of access on a particular server to access system information.

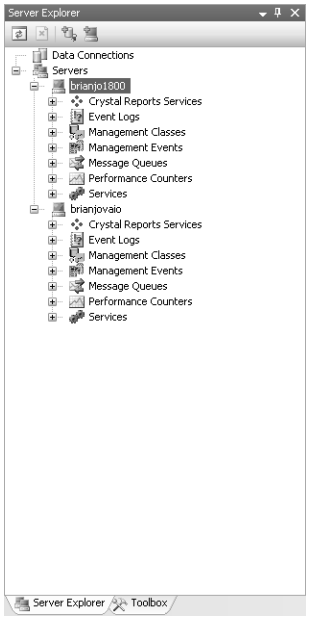


Figure 1-9 The Server Explorer window provides you with remote access to the machines you're working with.

Toolbox (Ctrl+Alt+X) The Visual Studio 2005 Toolbox window is used to hold the controls that you add to your Windows Forms and Web Forms applications. You can see the Common Controls in the Toolbox in Figure 1-10.

Command Window (Ctrl+Alt+A) The Command Window combines some of the best features of the Immediate window from Visual Basic 6 with the power of a command line. Chapter 2 covers the Command Window in detail. You use the Command Window to enter and execute named commands directly in Visual Studio 2005. A *named command* is essentially any IDE command that you can run through a menu, toolbar button, or shortcut. Many of the named commands in Visual Studio 2005 aren't mapped to a keystroke or available through a menu by default. The only way to access these commands without mapping them or adding them to a toolbar is to type them directly into the Command Window.

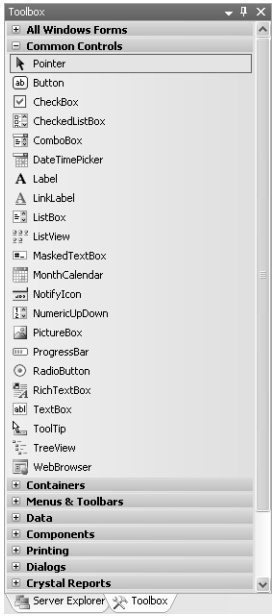


Figure 1-10 The Toolbox window gives you access to controls and code snippets.

In Visual Studio 2003, the Command Window had two modes of operation. In Command mode, the window acts as a command-line tool. In Immediate mode, the Command Window is used for debugging. In Immediate mode, you can execute statements, change variables and print their values, and evaluate expressions. In Visual Studio 2005, these two modes are broken into separate tool windows, making it very easy to switch between the two. To get to the Immediate mode tab from Command mode, type **immed**. This will open an Immediate Window tab, allowing you to execute debugging statements. To get back to the Command mode tab from Immediate mode, type **>cmd**.

Tip In Command mode, the > prompt will be visible on the line where you're typing your commands. Immediate mode shows no prompt.

Figure 1-11 shows the Command Window.

Figure 1-11 The Command Window in Visual Studio 2005 provides easy access to named commands in the IDE.

Macro Explorer (Alt+F8) The Macro Explorer window provides a view of the macro projects that are currently loaded in the IDE. Keep in mind that a macro project needs to be loaded in order for the macros in the project to be available for use or for editing in the Macros IDE.

When you record a temporary macro by pressing Ctrl+Shift+R, that macro becomes available through the MyMacros\Recording Module\TemporaryMacro item in Macro Explorer. You can rename the temporary macro to save it, or you can copy the code from the macro into another module in the Macros IDE. We'll discuss using recorded macros in more detail in Chapter 5. The Macro Explorer window is shown in Figure 1-12.

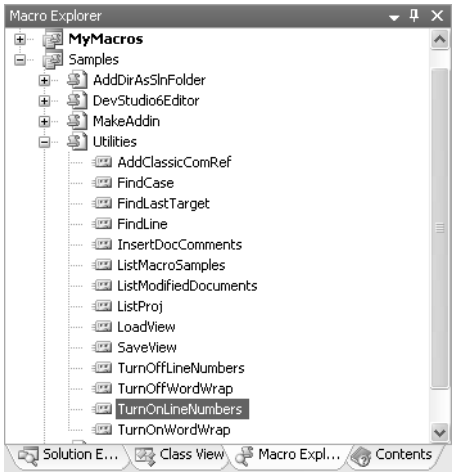


Figure 1-12 Macro Explorer gives you easy access to the macros available for use.

The IDE has a number of other important windows, which we'll talk about more fully in the next couple of chapters. Among these are the various debugging windows, the Help windows, and the Object Browser.

Visual Studio 2005 File Paths

In this section, we'll tell you a little bit about where Visual Studio 2005 places its important files. We'll cover this subject in more detail throughout the book, where it applies, but for now you should be aware of the locations of the files that you can manipulate to enhance the IDE and make automation a bit easier. The default base folder for the Visual Studio 2005 installation is \Program Files\Microsoft Visual Studio 8. Most of the folders we'll talk about in this section are subfolders under the Microsoft Visual Studio 8 folder (unless we provide the full path).

Installing Visual Studio 2005 also installs the .NET Framework SDK in the SDK\v2.0 subfolder. All the .NET Framework tools and samples are available in this folder, so it's a good place to start digging around if you're just getting to know .NET. Check out the StartHere.htm file in the SDK\v2.0 folder for the full story on the .NET Framework SDK.

The various languages that ship with Visual Studio 2005 all have their own subfolders that contain the project and solution templates for their respective project types. These folders are all named appropriately. Visual C++–specific files are found in VC, C#-specific files are in VC#, and Visual Basic 2005 files are in VB. In Visual Studio 2003, we used these folders to create and add custom projects to the various languages in the IDE. In Visual Studio 2005, you can store your templates in the My Documents\Visual Studio 2005 folder. This makes it much easier for you to find and extend these project types in addion to making it easier to run as a non-administrator on your workstation.

Tip You'll notice a file named Samples.vsmacros in the Common7\IDE folder. The sample macros for Visual Studio 2005 that run in your IDE are actually stored in your My Documents\ Visual Studio\ Projects\VSMacros80 folder. The version in the IDE folder is a backup copy. You can edit the Samples.vsmacros file in your My Documents\Visual Studio\Projects\ VSMacros80\Samples folder, but try to keep the version in your IDE folder clean. If you ever run into a macro corruption problem, you can usually copy the Samples.vsmacros file from your IDE folder to your VSMacros folder to get rid of the problem.

The IDE executable itself is Devenv.exe. This file is also available in the Common7\IDE subfolder. The IDE folder contains a number of subfolders that you'll be using throughout the book. These folders include the PublicAssemblies and PrivateAssemblies folders, which you'll use to add custom assemblies that are available to macros in the IDE. You'll use the HTML folder to customize the Start Page. The templates for the macro projects are stored in the MacroProjectItems and MacroProjects folders. Generic item templates (those not associated with a particular project type) are stored in the NewFileItems and NewScriptItems folders.

Adding an IDE Folder Shortcut to Your Tools Menu

If you do a lot of extensibility work, you might want to add a shortcut to the IDE folder to your Visual Studio 2005 Tools menu. To do this, follow these steps:

1. Press Ctrl+Alt+A to open the Command Window, and then type **Tools.External Tools**, or click **External Tool** in the Tools menu. This will open the External Tools dialog box.
2. Click **Add** to add a new tool to the menu, and type **IDE Folder** as the Title.
3. In the Command text box, type **Explorer.exe**.
4. In the Arguments text box, add the path to your Visual Studio 2005 IDE subfolder. (This is usually C:\Program Files\Microsoft Visual Studio 8\Common7\IDE.) Click **OK**.

If all of that works, your IDE folder should open when you choose **IDE Folder** from the **Tools** menu. We'll use the External Tools feature to create some more time-saving shortcuts later in the book.

If you do command-line builds or if you simply like to work from the command line, you'll want to set environmental variables for Visual Studio 2005 when you launch Cmd.exe. You have a couple of options for setting these variables. First, you can simply open the Start menu and choose the Visual Studio 2005 Command Prompt. You'll find that command prompt in the Visual Studio 2005 Tools folder, which is in the Microsoft Visual Studio 8 folder.

Tip We suggest pinning the Visual Studio 2005 Command Prompt link to the Start menu so that you'll have easy access to it as your primary command prompt.

The Visual Studio environmental variables are available in a file named vsvars32.bat, which is in the Common7\Tools subfolder. If you want access to these variables from every instance of Cmd.exe on your machine, you can add C:\Program Files\Microsoft Visual Studio 8\Common7\Tools to your system path. (Alternatively, you can copy this file to a folder in your path.) Then you can just type **vsvars32** from any command prompt and you'll have a Visual Studio 2005 working environment from your current command prompt.

You can take this one step further by creating a Command item on the Tools menu. You create a new menu item from the External Tools dialog box by clicking Add, making the Title of the new item **Command Prompt**, and making the Command item **cmd.exe**. If you want, you can set the Initial Directory box to **$(ProjectDir)**. Setting the Initial Directory to your project directory will open the command prompt to that directory. This can make it very convenient to work with your project files from the command line.

Finally, consider adding the Common7\IDE path to your system variables. The full path is C:\Program Files\Microsoft Visual Studio 8\Common7\IDE. This will make Devenv.exe available from any command prompt on your system. This path is added by the vsvars32.bat command, but sometimes you need access just to Devenv.exe.

Tip Consider using Devenv over Notepad when you're editing files for command-line builds. Even though you might not get access to the build and project facility without a solution, you still have access to your custom tools and to your macros.

Visual Studio 2005 Extensibility

Visual Studio 2005 builds on an extensibility model that was first developed for Visual C++ 5. In Visual Studio 2005, the *DTE* object—*DTE* in this context stands for Development Tools Extensibility—sits at the top level of an automation model that features nearly 200 objects.

The functionality provided by the DTE object model can be described as user-defined customization. The DTE API is available to developers who are programming macros, wizards, and add-ins. Even more functionality is exposed to commercial-language developers who use the Visual Studio SDK to add custom languages and designer support to the IDE.

The following sections describe the automation mechanisms available to developers who are customizing Visual Studio 2005.

Macros

The macros facility in Visual Studio 2005 provides programmers with easy access to the features available through the automation APIs. The macros facility features its own Macro Explorer tool window (described earlier in the chapter), an extremely powerful macro recording facility, and a full-blown Macros IDE that is itself extensible through the DTE object model and that allows you to debug your macros. We'll use macros to illustrate concepts relating to extensibility throughout the book. Chapter 5 covers creating and editing macros in the Macros IDE in detail.

Macros in Visual Studio 2005 are written in Visual Basic. Because the macros facility takes advantage of the .NET Framework, macro programmers have access to the entire .NET Framework base class library as well as to functionality exposed by custom assemblies built in any other .NET language.

To open the Macros IDE, just press Alt+F11 from inside Visual Studio. The Macros IDE will open in a new window. The first thing you'll notice about the Macros IDE is that its layout is extremely similar to the layout of Visual Studio 2005. In fact, you'll find that most of the features available to you as a Visual Studio 2005 programmer are available to you as a macro programmer.

Figure 1-13 shows the Macros IDE in its default layout. This layout features a Project Explorer window that shows all of your currently loaded macro projects.

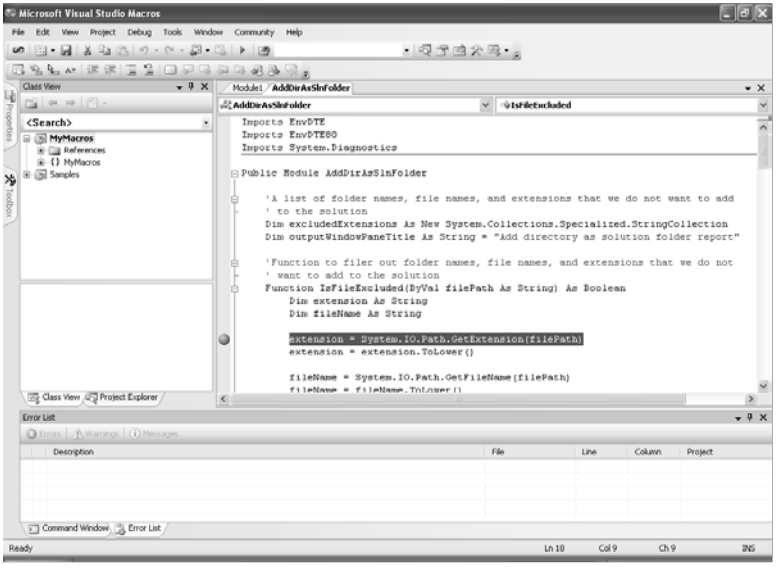


Figure 1-13 The Visual Studio 2005 Macros IDE

Add-Ins

Add-ins allow developers to create extensions to the Visual Studio IDE and to the Macros IDE. In general, compiled add-ins provide better performance than Visual Studio macros. Add-ins also provide functionality that integrates seamlessly into the environment. Independent software vendors (ISVs) and individual programmers can extend the IDE through add-ins in a way that makes the use of the add-in look just like a built-in part of the IDE.

Add-ins can be written in any .NET language, or they can be written as native COM components in unmanaged Visual C++. Add-ins are required to implement the *IDTExtensibility2* interface. Most of the add-in samples in this book will be shown in Visual C#.

Microsoft makes available a number of add-in samples that you can use to explore the extensibility object model or simply to add functionality to your Visual Studio 2005 IDE. Samples and more information are available at *http://msdn.microsoft.com/vstudio/extend/*. We'll use a few of these add-ins in the early chapters of the book to add specific features to Visual Studio. In Chapter 6, we'll provide all the details you need to build your own custom add-ins.

Wizards

Visual Studio wizards are similar to add-ins, but they are created using the *IDTWizard* interface. Wizards are fairly simple constructions that are designed to take a user step by step through a specific task.

Wizards are used in Visual Studio for a variety of purposes. Project wizards help get you started on a particular type of Visual Studio project, as shown in Figure 1-14. Other wizards in the IDE walk you through adding code to an existing project.

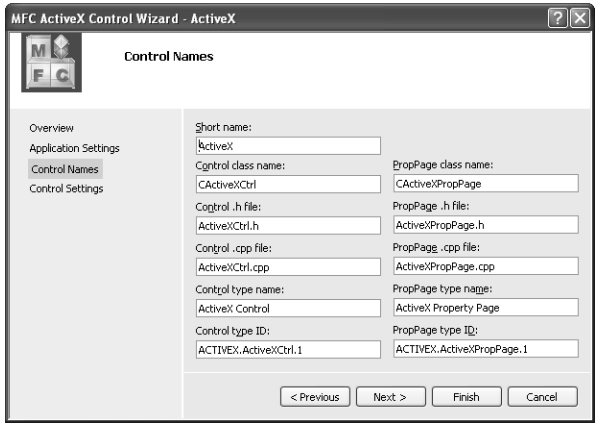


Figure 1-14 The MFC ActiveX® Control Wizard provides a starting point for a project.

Starter Kits

Starter kits are a new type of project template that lets you take an existing project and then supply it to others as an installable package. In Chapter 4, we'll show you how to create your own starter kits so that you can share extendable projects with the developer community.

The Visual Studio SDK

The Visual Studio SDK provides hooks into the IDE that allow developers to add their own languages and designers to Visual Studio. Developers can use these hooks to integrate new .NET languages and high-end tools into the IDE. Because developing with the SDK is fairly complicated, we didn't think that we could do it justice in this edition of the book. You can find out more about it through the Visual Studio Developer Center at *http://msdn.microsoft.com/vstudio*.

Looking Ahead

In Chapter 2, we'll continue our discussion of Visual Studio 2005, focusing on the project management facilities of the IDE. The chapter will provide some insight into one of the great strengths of Visual Studio 2005: the ability to host projects based on different programming languages in a single solution.

Chapter 2

Project Management in Visual Studio 2005

In this chapter:

Nearly everything you do in Microsoft® Visual Studio® 2005 revolves around solutions and projects. In this chapter, we'll talk about solutions and projects in detail and give you a good understanding of what those terms really mean. We'll also describe project management in Visual Studio 2005 and explain how you can organize your software projects to maximize the features of the integrated development environment (IDE).

Overview of Solutions and Projects

Managing complex software projects can be a difficult and messy affair. Visual Studio 2005 helps by organizing programming projects as solutions (groups of projects) and projects and by handling references to assemblies and to components outside this structure. This organization and reference feature helps promote code reuse by allowing you to take advantage of related projects, existing assemblies, COM components, and source code. The easiest way to reuse Microsoft .NET code is through references to assemblies in your projects and solutions.

Important Visual Studio 2005 organizes software projects on two conceptual levels. *Solutions* contain projects and solution items. *Projects* contain the source files that are compiled into executables and assemblies.

The most important tool for project management in Visual Studio 2005 is Solution Explorer (Ctrl+Alt+L), shown in Figure 2-1. Solution Explorer uses a tree-view window to provide access to all the projects and files that are part of the currently open solution. Visual Studio 2005

can host one solution at a time, but you can run multiple instances of Visual Studio if you want to work with multiple solutions concurrently.

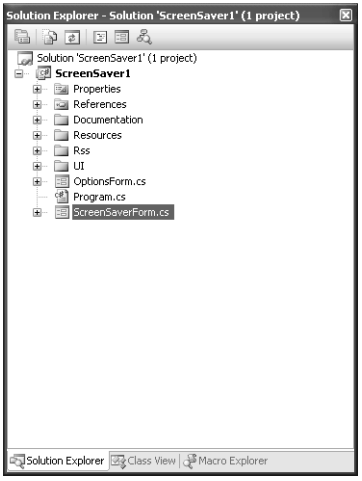


Figure 2-1 Solutions act as containers for projects and solution items.

Most new projects in Visual Studio 2005 are created using a template developed by a language integrator. For example, Visual Studio 2005 ships with support for Microsoft Visual Basic®, Visual C#®, Visual J#®, and Visual C++®. Each of these languages features a number of project types that programmers can choose from when creating a new project. A new project is created as part of a new solution by default. You can also add projects to existing solutions.

For Windows Forms applications and unmanaged Microsoft Windows®–based applications, the solution file for a project is by default stored in the same folder as the project. For Web applications, solution files are typically stored in a folder in the Visual Studio Projects folder in your My Documents folder and point to the Web server that's hosting the application.

A single project can be a member of many different solutions. Because it's so easy to reorganize your projects in Visual Studio 2005, you should feel free to create your initial projects with default solutions. Later on, you can move your projects around and add them to new solutions if you want.

Understanding Solutions

In Visual Studio 2005, a solution is a thin wrapper that contains a project or a number of projects. In earlier versions of Visual Studio, every project was part of a solution by default. In Visual Studio 2005, some project types can create a temporary solution for you. The

solution concept is important because much of what you can do in Visual Studio 2005 revolves around accessing functionality that's exposed in different projects.

Solution Items and Miscellaneous Files

Solutions can contain solution items and miscellaneous files in addition to projects. Solutions can also contain Solution Folders, which allow you to group related projects. Solution items can consist of HTML files, bitmaps, icons, XML files, templates, schemas, and others. Miscellaneous files can be a bit of a mystery. First of all, you need to make the Miscellaneous Files folder visible in Solution Explorer to take advantage of these kinds of files. To see this folder, select the Show Miscellaneous Files In Solution Explorer check box on the Documents page in the Environment folder of the Options dialog box, as shown in Figure 2-2. Keep in mind that you won't see the Miscellaneous Files folder until you open a non-project item in the IDE by using File.OpenFile.

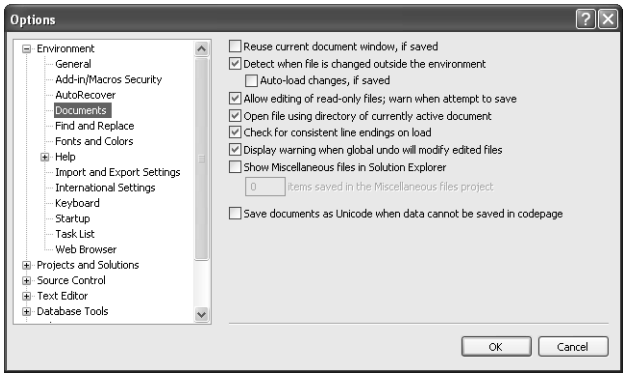


Figure 2-2 You can enable the Miscellaneous Files folder in the Options dialog box.

Miscellaneous files are files that you might open in the IDE for reference purposes—for example, if you want to review some code in a listing that you don't want to make part of your project. Opening such a file in the IDE without importing it into your solution automatically places the file into the Miscellaneous Files folder. The linked file is aggregated into the Miscellaneous Files folder in a solution.

Keep in mind that any file you open from Visual Studio 2005 gets a link in the Miscellaneous Files folder. This folder persists your items between sessions if you set Miscellaneous Files Project Saves Last to five items or so. This means that you can open specifications, schedules, and notes and have those files at your fingertips every time you open your project, as shown in Figure 2-3.

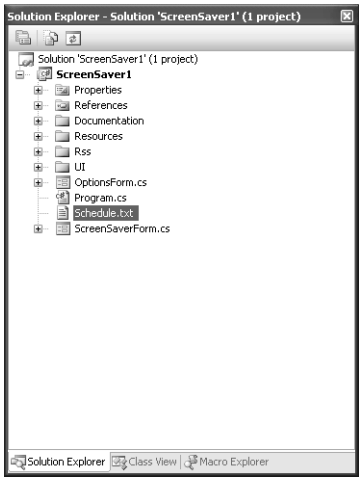


Figure 2-3 You can use the Miscellaneous Files folder to store links to documents that relate to your projects.

Solution Properties

The Solution Property Pages dialog box gives you easy access to the settings that apply to an entire solution. Among the options that you can control are the startup project or projects for your solution, the locations for files and symbols used for debugging, and the configuration settings that apply to the different projects in your solution.

To get to the Solution Property Pages dialog box, make sure that the solution name is selected in Solution Explorer, press Ctrl+Alt+A, and type **Project.Properties** in the Command Window. Another way is to right-click the solution and choose Properties. Most of the major programming projects in Visual Studio 2005 present you with the Property Pages dialog box, shown in Figure 2-4.

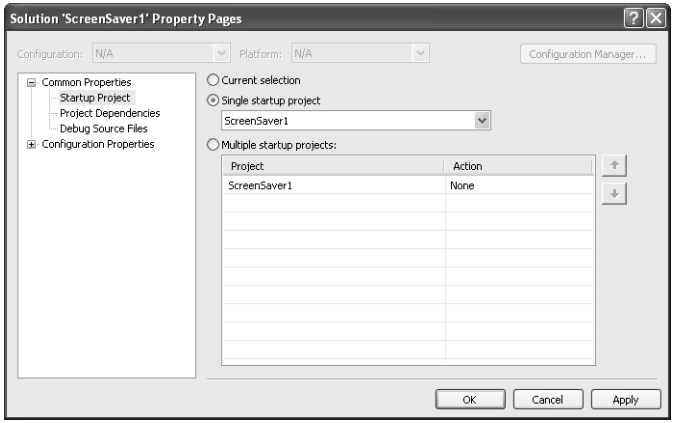


Figure 2-4 The Solution Property Pages dialog box gives you access to solution settings.

Common Properties

Clicking the Common Properties folder in the folder pane on the left exposes a number of options. The first option is Startup Project. In multiple-project solutions, you can select the project that launches when the solution is run from the Debug menu. You'll most often set this option on the fly by right-clicking a project name in Solution Explorer and then choosing Set As StartUp Project from the project shortcut menu.

If you want to run more than one project when you choose Start or Start Without Debugging from the Debug menu, select the Multiple Startup Projects option. Selecting this option lets you select the behavior of each of the projects in your solution when you invoke *Debug.Start* or *Debug.StartWithoutDebugging*. You can select Start, Start Without Debugging, or None. You can use the Move Up and Move Down buttons to the right of the list of projects to set the order in which the programs are started.

In a number of cases, running multiple projects concurrently might be useful. You might want to test some interprocess communication features between various assemblies in your solution. You might use a second project to do some profiling or instrumentation. Another use might be to run a utility that takes control of another assembly for automated testing purposes.

The second option in the Common Properties folder is Project Dependencies. When some assemblies in a solution depend on others in the same solution, the build order for the different projects in the solution is critical. The Project Dependencies settings let you specify which projects need to be built before others to get the entire solution up and running.

The last option in the Common Properties folder lets you set file paths for source files and debug symbols that might come up in your application. These settings allow you to step into the source code for libraries that are referenced by your projects but aren't part of your project. If you're debugging a project that's referencing a debug version of a .NET assembly, Visual Studio 2005 is usually able to find the source for the assembly if it's available. If the source is stored in a different location, you can specify the location of the source files and the debug symbols so that you can debug into that source.

Configuration Properties

Solutions can have multiple configurations that give you quick access to preset options that are related to your solution. The Debug and Release configurations are available to new projects by default, but you can create your own configurations by using Configuration Manager, which is accessible from the Solution Property Pages dialog box or from the Build menu (Build.ConfigurationManager).

Visual Studio 2005 offers two types of configurations: solution configurations and project configurations. Solution configurations are for configuring different build setups within a particular solution. For example, you can create and save a specific solution that allows you to select a different configuration for each project in your solution.

The second type of configuration is the project configuration. We'll discuss custom project configurations in detail later in the chapter, but for now, consider how different projects might relate to one another in a solution. Project configurations let you change some very specific build characteristics. These characteristics include code optimizations, debugging switches, and even the location of the project's compiled files. If you have five projects with different custom settings in a single solution, you should use custom solution configurations to save and manage different build options for your assemblies.

Solution and Solution User Options Files

The solution source .sln file is a plain-text document that describes the solution. The solution file contains links to the projects contained in the solution. It also contains version information about the format of the solution file itself.

Important Once you convert a file to Visual Studio 2005, you can no longer open it in earlier versions of Visual Studio.

The .sln file also contains information on the various configurations that have been set up in the solution. Information about the different solution configurations is stored in this file, along with information about how the different project configurations are organized in those solution configurations.

If you take a look at an .sln file in which solution items have been enabled and added, you'll notice that there's no information about these files. Solution items are considered user items, so links to these files are stored in the solution user options (.suo) file. If you pass a folder containing an .sln and an .suo file to another user on another machine, much of the information in the .suo file will become useless to the second user and will be ignored.

Some important items are stored in the .suo file that you can share with another person. Breakpoints that you set in your solution are stored in the .suo file, as are tasks that have been added to the Task List window. If you want to share that information with the person you're sharing the solution with, you should be sure to keep the .suo file in the same folder as the .sln file. If you don't need to share such information, we recommend deleting the .suo file because that file can contain personal and confidential data such as the paths to network shares and even your e-mail alias.

Projects

Projects are the second type of container used in Visual Studio. Projects are used to maintain the source files associated with individual assemblies, Web sites and services, and applications. As with solutions, Solution Explorer is the primary tool for managing projects in Visual Studio.

Project Items

Projects in Visual Studio 2005 consist primarily of file items. These items can be links to files or source files in the same folder as the project file. Whether an item is a link or an actual file depends on the type of project that you're working with. The files associated with Visual C++ projects are links displayed in Solution Explorer. These files are usually in the same folder as the projects. Deleting a link to a file in a Visual C++ project doesn't necessarily delete the file that's opened by the link. It's a rather fine distinction, but if you've ever moved a Visual C++ project and found yourself missing a project file, it might be that the file existed outside the project folder. In Visual Studio 2005, Show All Files now works in Visual C++ to show you all the files located in the physical directory.

.NET Windows projects can consist of a mix of links and actual file items. Web sites are generally contained in a solution but don't have a project file. Table 2-1 shows the possible relationships between project type and file items in Visual Studio 2005.

Table 2-1 Project Items in Visual Studio 2005

Project Type	Associated Items
Visual C++	Links to items
Web site	Items in the Web folder
Visual Basic	Links and actual items
Visual C#	Links and actual items
Visual J#	Links and actual items

If you take a look at Solution Explorer for a Windows application written in Visual C# or Visual Basic, you can see the mix of project items and file structure items by using *Project.ShowAllFiles*. Files that are part of the project will appear normally. Files that are not part of the project but are in the project folder will appear slightly grayed. The *ShowAllFiles* command is available from the Project menu and through toolbar buttons in Solution Explorer. In addition, you'll see a number of files that are kept hidden from the user by default. These hidden files include some types of configuration files and the code-behind files used in ASP.NET applications. In Figure 2-5, most of the items in the project shown are project items.

In addition to the items, a project file stores the configuration metadata associated with the project. Information stored includes configuration data that you specify in the IDE as well as build and debugging data. The nature of this data differs from project type to project type. The compilers for the different languages are written by different teams, so the available options differ from language to language.

Figure 2-5 Links and files in a Visual C# solution

Project Properties

You set the options for a project in the Properties window. (In Solution Explorer, right-click a project and choose Properties from the shortcut menu.) The Properties window contains options that you would otherwise have to specify at a command prompt when compiling a project; these settings match particular command-line options.

Considering the four major languages that ship with Visual Studio and the different types of projects that you can create, there are quite a few compiler options. This is where project configuration in Visual Studio becomes fun. By creating custom project configurations, you can try out many different types of builds and save those configurations for future use and reference.

Saving a Custom Configuration

You can access the Configuration Manager dialog box by entering **Build.ConfigurationManager** in the Command Window. To create a new project configuration, click the drop-down button adjacent to the desired project in the Configuration column of the Project Contexts grid and then click New. The New Project Configuration dialog box appears (shown in Figure 2-6).

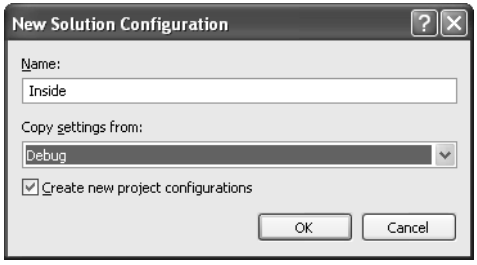


Figure 2-6 The New Project Configuration dialog box

Give your new configuration a name, and set the base settings for the configuration by selecting an existing configuration from the Copy Settings From drop-down list. As with the New Solution Configuration dialog box, you can create a new solution configuration automatically to match your new project configuration by selecting the Create New Project Configurations check box. At this point, you should be ready to experiment with some settings in your project. Just create a new test configuration that you can experiment with and leave all the default settings in the two default configurations.

Properties for managed applications written in Visual Basic, Visual C#, and Visual J# all display somewhat similar layouts in the Properties window. When creating a Web application, you can change project settings through that project's Properties dialog box.

Figure 2-7 shows the Application tab for a Visual Basic Windows application.

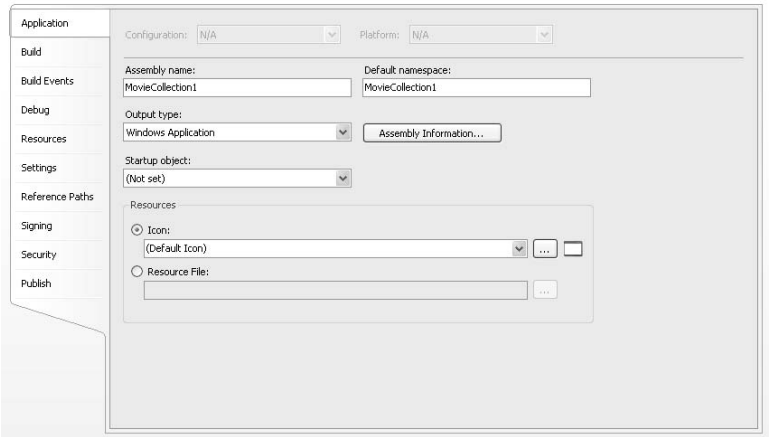


Figure 2-7 The Application tab for a Visual Basic Windows application

Configuration Properties

The Configuration list for a project can be found on a number of the tabs in the Properties window. You can use this drop-down list to experiment with settings and save them as separate build types. You can easily create and save new build types for almost any kind of Visual Studio 2005 project and compile them from inside the IDE.

In this section, we'll point out a few of the important settings in the Properties window for a Visual C# project. You can get to most of these settings in a Visual Basic project, as well.

Figure 2-8 shows the Build page from the Configuration Properties folder for a Visual C# project. You can save any of these settings to a custom build type. One of the most useful settings for configuring a custom build type is the output path. The default output path for a Visual C# Debug build is \bin\Debug\. The release build is \bin\Release\ by default. When you create a custom build type, you get one of these two paths, depending on which type of build you get your initial settings from. If you're creating a custom build, it might make

sense to copy the output of that build to a new folder so that you can compare the output assemblies. For cases like this, you can create a new build path to match your build name. For example, if you have a build named DebugOverflow (to indicate that you've enabled overflow checks for this build type), you can change the output to \bin\DebugOverflow.

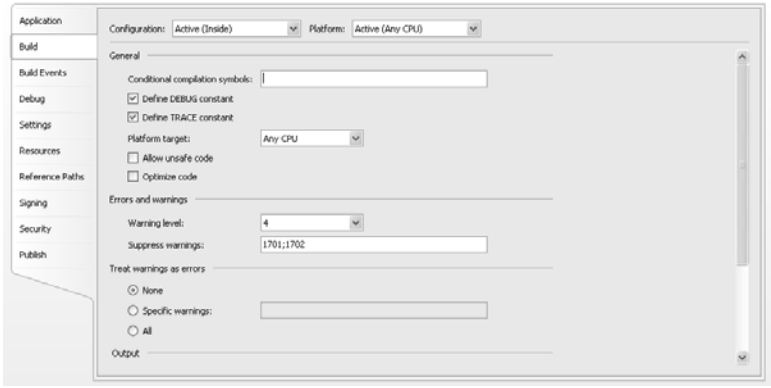


Figure 2-8 The Build page for a Visual C# Windows Forms application

The Debug page, shown in Figure 2-9, can be especially useful when you're building class library, Windows, and Web services projects. You can experiment with a lot of settings on this page, but one of the most useful to our discussion is the Start Action option. Using different build types, you can specify particular URLs that you want to test your Web service against. You can use the Start Action option in the same way to test your libraries. It lets you easily debug your service or library against a number of test applications.

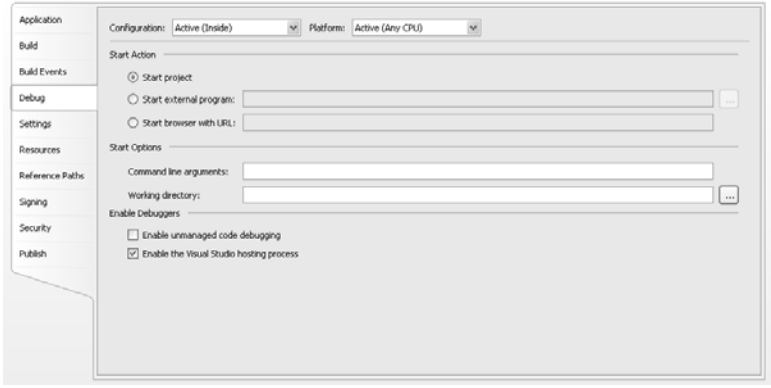


Figure 2-9 The Debugging page for a Web application

Visual C++ Projects

The Property Pages dialog box for Visual C++ projects has a huge number of settings because of the large number of compile and link options available. The custom build options that

we've talked about in this chapter apply to Visual C++ as well. In fact, because of the many properties available, you should find custom settings for unmanaged projects very useful, especially in testing and teaching situations.

Figure 2-10 shows the Property Pages dialog box for a Visual C++ Win32® project.

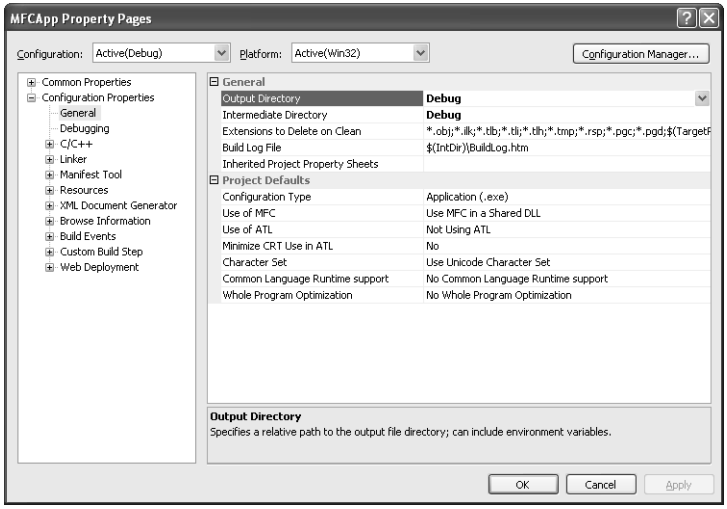


Figure 2-10 A custom configuration in Visual C++

The Property Pages dialog box for a Visual C++ project has a number of subfolders under the Configuration Properties folder. Table 2-2 contains a list of some of these folders and the general property types that you can set from each. If you're an experienced Visual C++ programmer, you'll find most of these settings fairly straightforward.

Table 2-2 Selected Configuration Properties Subfolders in Visual C++

Subfolder	Properties
General	Specify output directories, log options, MFC/ATL options, CLR support (/CLR).
Debugging	Specify which debugger you want to use from the IDE. These include the local debugger, remote debugger, or other debugger.
C/C++	Compiler options, preprocessor definitions, paths to some output files, and command-line compile options.
Linker	Link options, debug options, and command-line link options.
Resources	Resource file name and path, culture, and resource compiler command line.
Browse Information	Options relating to BSCMAKE (browser files).
Build Events	Commands that you can run during the build process.

Table 2-2 Selected Configuration Properties Subfolders in Visual C++

Subfolder	Properties
Custom Build Step	Properties for configuring an additional task you specify when building a file or a project. For example, you might pass an input file to a tool that returns an output file.
Web Deployment	Specifies how a Web deployment tool will install your application.

The Build Events node allows you to do a few interesting things with your custom builds. You can see the Post-Build Event page in Figure 2-11. If you're working with multiple projects and builds, you can use the Build Events folder to run applications and scripts during your build process. In this case, we've added a call to Regsvr32.exe as the command line for the Post-Build Event in the project. After this project is built under this configuration, the target file is registered with Windows.

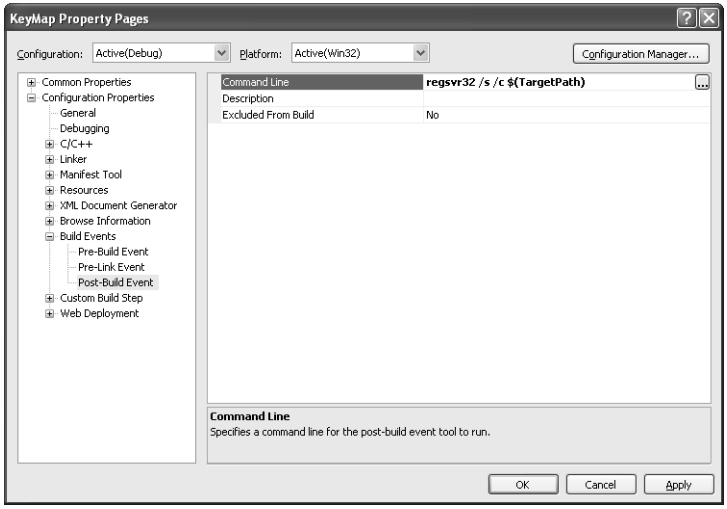


Figure 2-11 Build events let you run applications during your build process

Project Source Files

Project source files have different extensions based on the language specific to the project. Table 2-3 lists the project types and extensions that they hold.

Table 2-3 Project Types and Extensions

Project Type	Extension
Visual Basic	.vbproj
Visual C#	.csproj
Visual C++	.vcproj

Table 2-3 Project Types and Extensions

Project Type	Extension
Visual J#	.vjproj
Deployment	.vdproj

Visual Basic, Visual C#, and Visual J# projects also contain user option files. These files take the form *ProjectName.ProjectExt*.user. A Visual Basic user options file has the extension .vbroj.user. These project user files are in XML and contain information specific to the custom builds that you've created. Unlike the binary .suo file, the .user files are intrinsic to custom build types and should usually be kept with a project.

Project Dependencies

If you're building complex solutions that contain a number of assemblies with interproject dependencies, you can take advantage of Solution Explorer to help you manage these dependencies. Solution Explorer makes it really easy to add file, project, and Web references to your projects. For solutions with dependencies between projects, you'll want to use project references.

To add a project reference, open the Add Reference dialog box by selecting a project in Solution Explorer and typing **Project.AddReference** in the Command Window. On the Projects tab, you'll see a list of the projects in your solution, as shown in Figure 2-12.

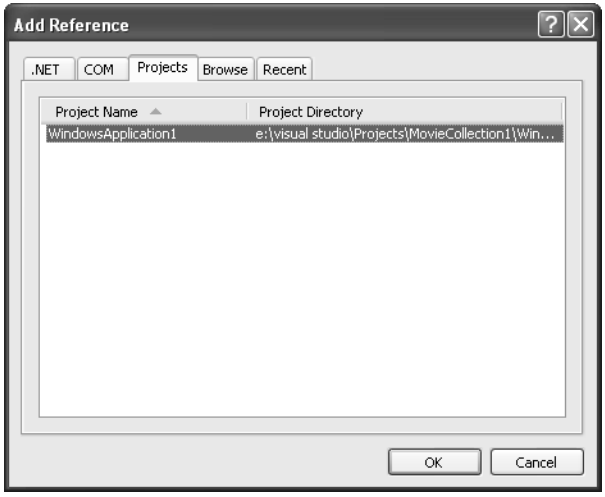


Figure 2-12 Adding a project reference to a project in a solution

After you add a project reference, the functionality available from the referenced project becomes available to the project adding the reference. At this point, build order becomes important because the referenced assembly must be built before the project that references

it. To help you manage dependencies such as this, Visual Studio provides a Project Dependencies dialog box (*Projects.ProjectDependencies*), as shown in Figure 2-13. This dialog box lets you specify a dependency, and it then changes the build order of affected projects in a solution accordingly.

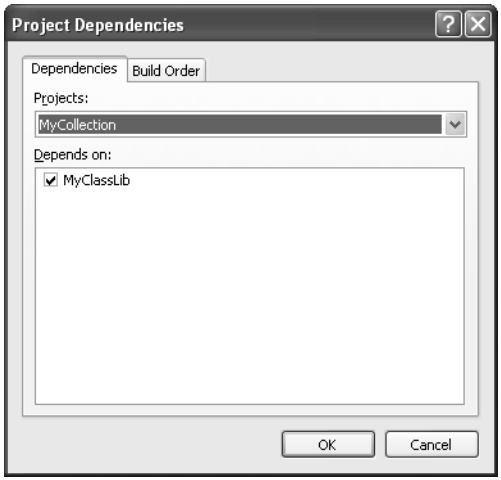


Figure 2-13 Configuring build dependencies for a project

Building Projects and Solutions

Once the projects, custom build configurations, and references are set in a solution, you can begin to work out the build scenarios that you want to run with the different configurations. To specify which projects in the solution should be built, you can use the Configuration Manager dialog box (shown in Figure 2-14). You can easily exclude projects that might give you problems, or you can simply save some time when you want to concentrate on a specific build in a solution.

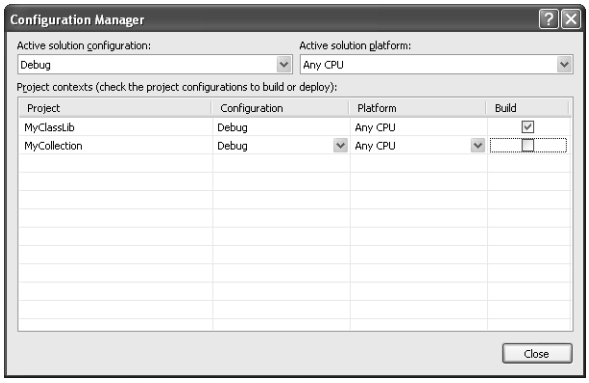


Figure 2-14 Determining which project to build for a given build configuration

There's one more powerful build dialog box you can use to build multiple configurations at one time. The Batch Build dialog box (*Project.BatchBuild*), shown in Figure 2-15, is helpful for building multiple projects with different configurations, but it's not available for every application type.

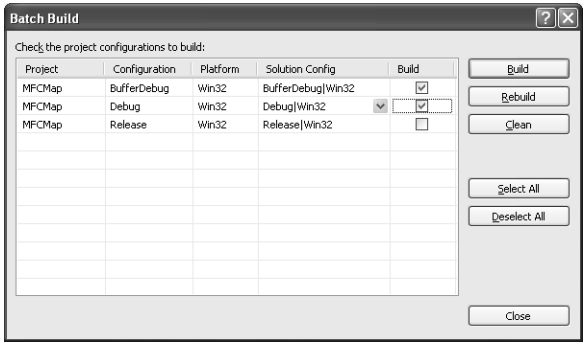


Figure 2-15 The Batch Build dialog box

The Batch Build dialog box lets you perform a number of important actions, including building, rebuilding, and cleaning your projects. Clicking the Build button initiates an incremental build for projects that are configured for such a build. The Rebuild button initiates a Rebuild All for all the selected projects. Clicking the Clean button deletes the files that are output by a build so that you can restart the build cleanly or share your projects without unnecessary bulk.

Looking Ahead

Although the project management facilities in Visual Studio are formidable, the usability of the IDE for developers also relies on the editor features in the IDE. In Chapter 3, we'll discuss the code editor in detail and discuss techniques that can help you become more productive as you write code.

Chapter 3

The Visual Studio Editor

In this chapter:

The editor is the heart of any development environment. In this chapter, we'll take a close look at the editing tools built into Microsoft® Visual Studio®. Whether you're a new or experienced programmer, the information in this chapter will help you become even more productive in Visual Studio 2005. We'll show you how to access editor features that make your job easier, and we'll describe some of the features of the integrated development environment (IDE) that make working in Visual Studio 2005 a real pleasure.

Documents in the IDE

In Visual Studio, everything you do revolves around the solution and the projects in the solution. In that way, Visual Studio becomes your project management tool. What you're managing, for the most part, are the source documents that comprise your solution. To create and edit these documents, you use the Code Editor and the designers in the IDE. The source files you're editing show up in windows that open to the center of the IDE and become part of the tabbed view. The windows that contain these files are known collectively as *document windows*, and they can be designers, editors, a Web browser, and Help windows.

All these features can be accessed by using named commands either from the Visual Studio Command Window (Ctrl+Alt+A) or through menu commands or keyboard shortcuts. Master these commands and you master Visual Studio 2005.

Dockable Tool Windows

Not all the tabbed windows in the IDE are document windows. You can add a tool window to the tabbed windows at the center of the IDE by selecting the window and toggling off the window's Dockable value on the Window menu. The Object Browser window (Ctrl+Alt+J) is undocked by default, making it a tabbed window in the IDE. The benefit of adding a tool window to the set of tabbed windows at the center of the IDE is that you can display a large amount of information at once. Alternatively, you can undock a tool window by dragging it away from the edge where it's docked and leaving its Dockable setting on, essentially making the window a floating window. This technique is especially handy if you're working with multiple monitors.

It's All About Text

The place where you write your code in Visual Studio goes by one of two names, depending on the context of the file being edited. When you're working on a file that's been saved as a programming language type recognized by Visual Studio, the editor you're working in is called the *Code Editor*. The functionality that you'll find attached to the Code Editor will depend on how your language was integrated into the IDE. When you're working on a text file or a file type that's not been recognized by the IDE, the editor is called the *Text Editor*. This editor has less functionality than the Code Editor, but it's still fairly powerful. It's important to note that you can run macros in either editor, although the Code Editor hosts a much larger feature set. For the most part, we'll refer to these editors collectively as the Code Editor, but we'll make a distinction where appropriate.

Figure 3-1 shows the various parts of the Code Editor in the IDE. Take a look at the names used in the figure. The parts are probably familiar to you from a usage standpoint, but you might not be aware of their names. Depending on which language you're using, you might find some slight naming differences in the Code Editor, but the functionality is fairly consistent between languages.

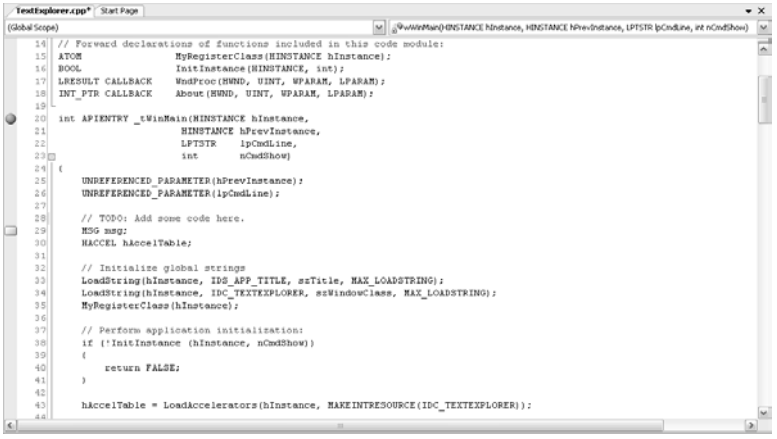


Figure 3-1 The parts of the Code Editor window

The Code Editor is where you type in code and text. You can click and drag just above the scroll bar on the right side of the Code Editor to break the view into two separate Code Panes. By using multiple Code Panes with a single file, you can look at different parts of your code concurrently. You can also open a new window on your code by typing **Window.NewWindow** from the Command Window.

The Navigation Bar contains the two boxes at the top of the Code Editor. In Microsoft Visual C#®, these are the Types and the Members drop-down lists. In Microsoft Visual Basic®, these boxes are called the Class Name and Method Name combo boxes. In Microsoft Visual C++®, they are called the Scopes and Members boxes, and in a Web project, these are called the Object and Event boxes. You can use the Navigation Bar to jump quickly to various parts of your code. In Visual Basic, you can also use these boxes to add methods to the current source file.

The light vertical line to the left of the code is the outlining indicator. By clicking the + and − boxes along this line, you can show and hide blocks of code, respectively, within a source file.

Lines of code that changed in the current editing sessions are marked in yellow and green in the Indicator Margin. Changes that have been saved are marked in green, and changes that have yet to be saved are marked in yellow.

The area between the rightmost part of the outlining indicator and the Indicator Margin is the Selector Margin. Clicking in the Selector Margin selects the adjacent line of code. When your mouse pointer is in the area of the Selector Margin, it changes from an arrow pointing to the top left to one that's pointing to the top right. By clicking and dragging down or up in this area, you can select complete blocks of code, as shown in Figure 3-2. The benefit is that you end up selecting the same amount of white space in each line of code, giving you a nice, clean block.

```
172  // Message handler for about box.
173  INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
174  {
175      UNREFERENCED_PARAMETER(lParam);
176      switch (message)
177      {
178      case WM_INITDIALOG:
179          return (INT_PTR)TRUE;
180
181      case WM_COMMAND:
182          if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
183          {
184              EndDialog(hDlg, LOWORD(wParam));
185              return (INT_PTR)TRUE;
186          }
187          break;
188      }
189      return (INT_PTR)FALSE;
190  }
191
```

Figure 3-2 Selecting text by using the Selector Margin

The Indicator Margin is a tool that serves many purposes. It's used to set and delete breakpoints in your code, to indicate bookmarks in code, and to hold Task List shortcuts. During debugging, you'll see an indicator in this margin. When a breakpoint is hit, the breakpoint indicator will contain a yellow arrow that points to the current line of code. This line of code is highlighted in yellow by default. As you step through the code, the yellow indicator shows you where you are in the code, and that statement is highlighted in

the Code Editor. You'll also see a Current Line indicator in this margin as you search and navigate your source code.

Notice the tab at the top of the Code Editor window. These tabs let you navigate easily between multiple source files and forms in your project. If you prefer working with multiple document interface (MDI) windows, such as the ones in Visual Studio 6, you can turn off the tabs in the Options dialog box.

Now that we've reviewed the Code Editor window, let's take a look at the kinds of things we can do inside the Code Editor to make programming and editing tasks easier.

Typing and Shortcuts

Two of the most important characteristics of a good editor are efficient typing and text manipulation. If you're new to programming, you might not be aware of the ferocious battles being fought in chat rooms, newsgroups, and Web sites between factions of programmers who prefer one editor over another and who will argue incessantly about what makes their editor better than another.

So what is it about text editing that causes such a strong reaction among programmers? I think it has to do with the idea that programmers like to find the most efficient way to do anything, and if a specific editor allows them to accomplish their goals, they become very attached to that editor. A secondary reason is that it takes some time to master an editor, and, once a programmer does so, he's less likely to want to learn things all over again unless a better editor comes along.

The sections that follow are designed to show you how Visual Studio can work for a programmer who likes to keep her hands on the keyboard. I've noticed that many of the most productive programmers I work with rarely take their hands off the keyboard to perform routine tasks that less experienced programmers use the mouse to perform. The idea behind these shortcuts is to improve your speed in the IDE, and they take some time and practice to learn.

Common Editing Shortcuts

Applications written for Microsoft Windows® use a number of standard keyboard shortcuts that you're probably familiar with. These shortcuts are known as Common User Accessibility (CUA) shortcuts and are based on work done at IBM that has standardized shortcuts across a number of platforms. The biggest advantage of using this particular set of shortcuts is that once you learn them, you can apply them in almost any Windows application, including Microsoft Office. These shortcuts have also been labeled on a number of popular keyboards, including most of the Microsoft keyboards.

Tip For more information about Windows keyboard shortcuts, see the book *Microsoft Windows User Experience* (Microsoft Press®, 1999), which details how shortcuts such as these should be used in Windows applications.

The tables that follow group the common editing shortcuts for Visual Studio based on function and when you're likely to use them in an editing session. Table 3-1 lists the file shortcuts. You'll use these to open new files or existing files and to save files as you work.

Table 3-1 Common File Shortcuts

Command	Keystroke	Named Command
New	Ctrl+N	File.NewFile
Open	Ctrl+O	File.OpenFile
Save	Ctrl+S	File.SaveSelectedItems
Save All	Ctrl+Shift+S	File.SaveAll
Print	Ctrl+P	File.Print

You'll notice that you're presented with a New File dialog box when you try to create a new file in Visual Studio. This might take a little getting used to if you prefer to see a new text document appear immediately. By selecting a specific file type when you create the new file, you enable much of the functionality associated with a particular language before you save the file. You can save some time when creating a new file by using the Command Window and adding the name and extension of the file you want to create. For example, if you want to create a file named UserMotion.cpp, press Ctrl+Alt+A to open the Command Window and then type **File.NewFile UserMotion.cpp**. Later in this chapter, in the section "Using the Command Window," we'll show you how to alias commands such as this one so that you can easily create the files you use most often.

Navigating in a document by using keystrokes is one of those skills you tend to learn without actually picking up a book or reading an article. We'll review the common navigation and selection keys and shortcuts here. They are listed in Table 3-2. Notice that selection involves holding down the Shift key and that moving to a larger selection for a particular key usually involves holding down the Ctrl key.

Table 3-2 Common Navigation and Selection Shortcuts

Movement	Movement Keystroke(s)	Selection Keystroke
Character	Right Arrow	Shift+Right Arrow
	Left Arrow	Shift+Left Arrow
Word	Ctrl+Right Arrow	Ctrl+Shift+Right Arrow
	Ctrl+Left Arrow	Ctrl+Shift+Left Arrow
Line	End	Shift+End
	Home	Shift+Home
	Down Arrow	Shift+Down Arrow
	Up Arrow	Shift+Up Arrow

Table 3-2 Common Navigation and Selection Shortcuts

Movement	Movement Keystroke(s)	Selection Keystroke
Code Pane	Page Down	Shift+Page Down
	Page Up	Shift+Page Up
Document	Ctrl+End	Ctrl+Shift+End
	Ctrl+Home	Ctrl+Shift+Home

Once you've selected text, you can copy or cut it to the Clipboard, and then you can paste it back into the Code Editor. The common editing shortcuts are listed in Table 3-3.

Table 3-3 Common Editing Shortcuts

Command	Keystroke	Named Command
Cut	Ctrl+X	Edit.Cut
Copy	Ctrl+C	Edit.Copy
Paste	Ctrl+V	Edit.Paste
Undo	Ctrl+Z	Edit.Undo
Redo	Ctrl+Y	Edit.Redo
Select current word	Ctrl+W	Edit.SelectCurrentWord
Select all	Ctrl+A	Edit.SelectAll

Finally, let's take a look at the shortcuts that you can use to transpose letters, words, and lines. You can use the shortcuts shown in Table 3-4 to swap the position of two items in the Code Editor. For example, if the cursor is positioned before the letters *AB*, pressing Ctrl+T will cause the letters to switch their order to *BA*. Typing Ctrl+Shift+T with the cursor adjacent to or in the word *go* in the string *go boldly* will result in a transposition to *boldly go*. The most useful shortcut in this group runs the command *Edit.LineTranspose*. Using the shortcut Alt+Shift+T swaps the line where the cursor is located with the next line, making it really easy to move a line of code down the page.

Table 3-4 Transposition Shortcuts

Command	Keystroke	Named Command
Transpose character	Ctrl+T	Edit.CharTranspose
Transpose word	Ctrl+Shift+T	Edit.WordTranspose
Transpose line	Alt+Shift+T	Edit.LineTranspose

These shortcuts should provide you with the functionality you need to perform a fair number of editing tasks without the mouse if you choose to work that way. There's nothing

wrong with using the mouse for editing. It's not really much slower to use the mouse than to use shortcuts, but the extra second or two that it takes to go to the mouse can take you out of that creative groove you can get into when you're editing. For me, transitioning from using a mouse back to the keyboard takes a little more time than using a shortcut, so I try to use shortcuts whenever possible.

Custom Keyboard Shortcuts

Earlier we talked about toggling a window's Dockable state to add it to the center of the IDE. There's no shortcut assigned by default to the *Window.Dockable* command, but you might find that adding one would be handy for making a very data-heavy window easier to read.

To create a new shortcut in the IDE, press Ctrl+Alt+A to open the Command Window and type **Tools.Options**. This will bring up the Options dialog box (shown in Figure 3-3). Click the Keyboard item in the Environment folder to bring up the Keyboard page. This page lets you do a number of things with shortcuts in the IDE, such as create and edit shortcut keys, change the keyboard mapping scheme, and save a custom mapping scheme. The first time you add a custom shortcut to the IDE, you'll be prompted to save your mapping scheme with a custom name.

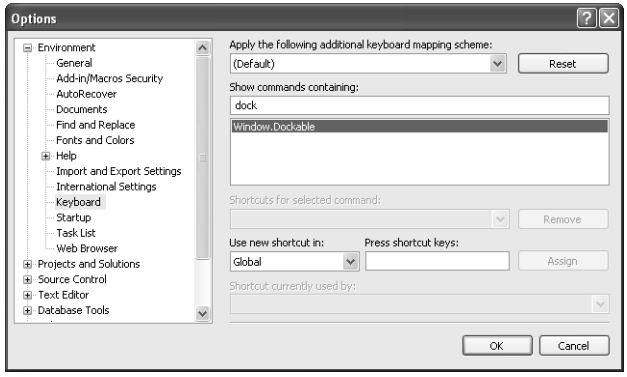


Figure 3-3 The Keyboard page of the Options dialog box

To find the command you want to assign the new shortcut to, type part of the command name in the Show Commands Containing box. In this case, type **dock**, and Window.Dockable will show up in the command list.

Here's the tricky part. Nearly every possible keystroke shortcut has been taken in Visual Studio. You can overwrite keystrokes that you think you'll never use, but that isn't always the most satisfactory solution. For one thing, if you go to work on a different machine and you haven't updated the shortcuts, you might end up keying the wrong command, which can be both annoying and potentially harmful to whatever you're typing at the moment. Your best bet is to find an available keystroke and take maximum advantage of it.

Visual Studio allows you to create keystroke sequence shortcuts. To start a sequence, you hold down the Ctrl key and press another key. The IDE then waits for another stroke to determine which command to execute. I've found that Ctrl+0 (Ctrl+zero) hasn't been taken in Visual Studio by default. So I can chord all my personal commands based on this key sequence and assign the second key sequence to one that matches the command I'm trying to execute. For the *Window.Dockable* command, I assign the sequence Ctrl+0, Ctrl+D (Ctrl+zero, Ctrl+D) by typing that combination in the Press Shortcut Key(s) box. Be sure to save the new shortcut by clicking the Assign button.

While I have the Options dialog box open, I can add a keystroke shortcut for the Options dialog box itself by typing **Tools.Options** in the Show Commands Containing box and assigning the keystroke Ctrl+0, Ctrl+O. Now I can open the Options dialog box quickly at any time to customize my IDE.

You can assign keystroke shortcuts to named commands in the IDE, to add-ins that you create or install, and to macros that you create and save.

Other Keyboard Schemes

In addition to the keyboard shortcuts mentioned here, Visual Studio 2005 provides a number of other keyboard shortcut mappings for you to choose from. To select a new scheme, go to the Options dialog box (Tools.Options) and look on the Keyboard page where we earlier assigned some keyboard shortcuts. At the top of the page, you'll see a drop-down list that allows you to choose from a number of different keyboard mappings. These mappings include Visual C++ 2.0, Visual Studio 6, Emacs, and Brief. To use one of these mappings, select the one you want from the list and click OK. These mapping schemes are designed to make it easy for programmers to move from their preferred editor to Visual Studio 2005. For more information on this feature, see the Shortcut Keys topic in Visual Studio Help.

Note Emacs and Brief support in Visual Studio 2005 are really more than just key bindings. They are emulations that offer special text selection and indenting behavior in the IDE.

Understanding Tabs and Code Formatting

Code formatting is another one of those issues that developers tend to feel strongly about. When it comes to code formatting, the bottom line for most organizations is that some sort of standard should exist. The formatting options for each of the languages supported in the Visual Studio IDE are set in the Text Editor node of the Options dialog box. When you set options for All Languages, as shown in Figure 3-4, you override the settings for each individual language listed in the Text Editor folder.

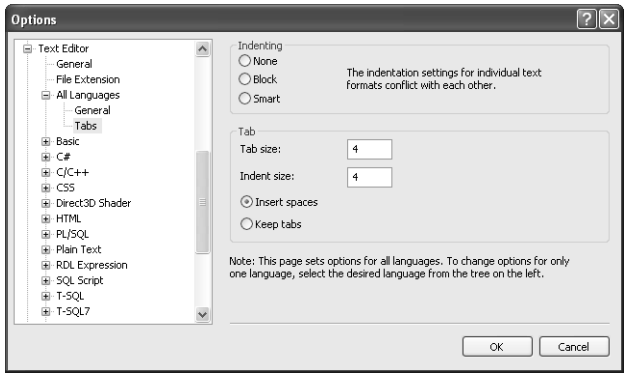


Figure 3-4 Setting global Tabs options

As you can see in the figure, you can set Indenting to None, Block, or Smart. The behavior of these options is determined by the language and the Tabs settings below them. When None is the selected Indenting type, pressing Enter at the end of a line will start the next line at the leftmost space in the Code Editor. Block indenting sets the indent to the same space as the first character in the current line. This is a common generic setting that lets you indent manually but doesn't force you to key a lot of extra tabs to get to where you want to be. The Smart setting applies an indent by context. For example, pressing Enter after an open brace ({) in C# will automatically indent the next line.

The choice between using spaces or tab characters for indenting is usually a matter of personal preference or of the coding standard you want to apply. If you prefer that all the code you deal with consists of spaces rather than tabs, you can set that option globally when you customize your IDE. If you prefer spaces to tabs, keep in mind that Visual C# specifies tabs for indentation by default. You can view the white space in the document by using the *Edit.ViewWhitespace* command (Ctrl+R, Ctrl+W). Figure 3-5 shows a document in which the white space is visible. If you use tab characters in your source code, they will show up as right arrows. If you use spaces in your code, a single dot will show up for every space.

If you want to convert existing files from tabs to spaces or vice versa, select the desired option on the Tabs page of the Options dialog box and then click OK to close the dialog box. Then simply press Ctrl+K, Ctrl+D to apply the new formatting to the whole document.

The Formatting page of the Options dialog box, shown in Figure 3-6, controls a number of characteristics of code typed into a Code Pane. This page is available for most of the major languages supported in the IDE, but under Basic, the VB Specific subpage handles the customizations. The page in the figure shows the C/C++ formatting options.

Different options are available for different languages. Notice the Indent Braces check box in Figure 3-6. In C++, this check box is clear by default. This setting forces curly braces that enclose functions to be indented by one tab or by four spaces, depending on how you have your spaces and tabs configured. You can change the formatting of an entire document by setting this preference and pressing the Ctrl+A then Ctrl+K, Ctrl+F keystrokes that we used earlier.

```
139
140  →  switch·(message)
141  →  {
142  →  case·WM_COMMAND:
143  →  →  wmId····=·LOWORD(wParam);
144  →  →  wmEvent·=·HIWORD(wParam);
145  →  →  //·Parse·the·menu·selections:
146  →  →  switch·(wmId)
147  →  →  {
148  →  →  case·IDM_ABOUT:
149  →  →  →  DialogBox(hInst,·MAKEINTRESOURCE(IDD_ABOUTBOX),·hWnd,·About);
150  →  →  →  break;
151  →  →  case·IDM_EXIT:
152  →  →  →  DestroyWindow(hWnd);
153  →  →  →  break;
154  →  →  default:
155  →  →  →  return·DefWindowProc(hWnd,·message,·wParam,·lParam);
156  →  →  }
157  →  →  break;
158  →  case·WM_PAINT:
159  →  →  hdc·=·BeginPaint(hWnd,·&ps);
160  →  →  //·TODO:·Add·any·drawing·code·here...
161  →  →  EndPaint(hWnd,·&ps);
162  →  →  break;
163  →  case·WM_DESTROY:
164  →  →  PostQuitMessage(0);
165  →  →  break;
166  →  default:
167  →  →  return·DefWindowProc(hWnd,·message,·wParam,·lParam);
168  →  }
169  →  return·0;
```

Figure 3-5 Displaying white space in the Code Editor

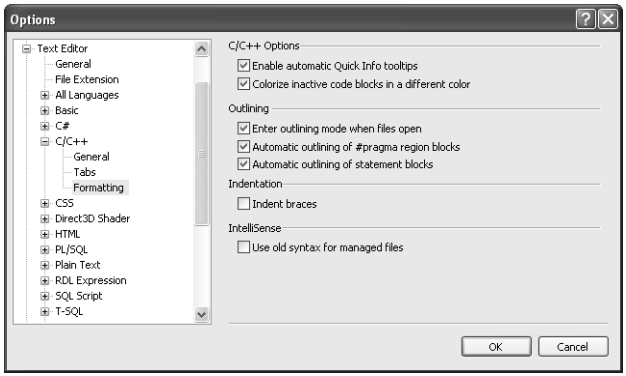


Figure 3-6 The C/C++ Formatting page

Other Editing Features in Visual Studio 2005

Editing in Visual Studio 2005 is much easier thanks to a number of new and updated features that have been added to the IDE since Visual Studio 2003. Among these are Code Snippets, refactoring, the Code Definition Window, and the Call Browser.

Code Snippets

Code Snippets is a feature added for Visual Basic, Visual C#, Microsoft Visual J#®, and XML that allows a programmer to insert commonly used snippets of code into a project. You can insert a snippet by right-clicking in the Code Editor in a supported project type and clicking Insert Snippet. Figure 3-7 shows you how this looks in Visual Basic. You can insert a small

amount of code or a significant block, depending on the functionality that you're adding. This can be a real timesaver as you work on your projects.

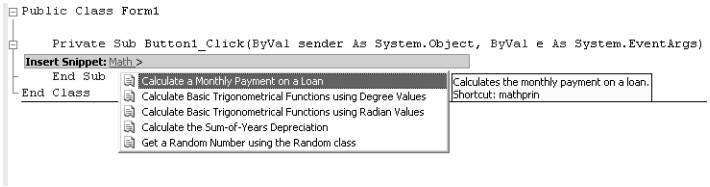


Figure 3-7 Inserting a code snippet into the Code Editor

Snippets are managed through the Code Snippets Manager. You can create your own snippets in XML. The Code Snippets Manager, available from the Tools menu, allows you to import XML files with your own code snippets.

Refactoring

Refactoring allows you to improve your code after you've written it by providing tools for automatically updating the structure of the code without making any changes to the functionality. This means that you can write a bit of code and then later use the tools in Visual Studio 2005 to change that code to improve the structure of your functions or to better encapsulate them for re-use. For example, by extracting an algorithm from a *button_click* event, you make it much easier to access the product of that algorithm from a different part of your application.

Code Definition Window

The Code Definition Window is a new tool in Visual Studio 2005 for Visual C++, Visual C#, and Visual J#. This window shows the definition of a selected symbol in the IDE. You can use the Code Definition Window to view the code definition file for the particular symbol if it's available. This window is dynamic, and it changes when you select various symbols in a source file. The Code Definition Window is available from the View menu or by using the View.CodeDefinitionWindow command in the Command Window.

Call Browser

The Call Browser is an interesting tool in that it lets you see calls to specific functions by displaying a *Callers Graph* that contains links to those places in the project from which the function is being called. This can be very useful when you want to see exactly who is calling your function. When a function is analyzed in the IDE, the title of the Call Browser window changes to Callers Graph and presents you with a tree that you can use to navigate your code.

Line Numbering and Outlining

None of what we're talking about in this chapter has anything to do with what happens when you build your applications. Depending on the language you use, compilers remove formatting and white space when a file is processed. At the editor level, however, even the small features provided in the IDE can have a profound effect on your productivity and comfort when you're working with code.

Line Numbering

You can set line numbering on the General page for any of the languages available in the Text Editor folder in the Options dialog box. You can set this option for any specific language, or you can set it for all languages. You can toggle this setting in the Options dialog box, but there is no named command associated with this setting.

To toggle this setting without opening the Options dialog box, you have to run a macro or an add-in to automate that functionality. Two of the macros that are part of the Samples macros set included with Visual Studio were designed to turn line numbering on and off. You can customize these macros yourself, or you can use them from the Command Window, from shortcuts, or by creating new menu commands or toolbar buttons.

The line-numbering macros are *Macros.Samples.Utilities.TurnOnLineNumbers* and *Macros.Samples.Utilities.TurnOffLineNumbers*. Either of these can be a finger buster to type into the Command Window, even with the aid of IntelliSense®, so we'll create an alias for each of these commands. An *alias* is a short command name that's used to represent a longer command in the Command Window. To create an alias for the *TurnOnLineNumbers* macro, open the Command Window by pressing Ctrl+Alt+A and type the following command:

```
>alias  lnon Macros.Samples.Utilities.TurnOnLineNumbers
```

Now when you type lnon, line numbering will be turned on (if it's currently off). To turn line numbering off, we'll create an alias for the *TurnOffLineNumbers* macro by typing the following into the Command Window:

```
>alias lnoff Macros.Samples.Utilities.TurnOffLineNumbers
```

The *TurnOffLineNumbers* macro is now mapped to lnoff.

Suppose you now want to map these macros to keyboard shortcuts. That's not a problem—you can just search for the word *Numbers* on the Keyboard page in the Environment folder of the Options dialog box, as we did earlier. Select the macro that you want to map to a keystroke, type your keystroke, and then click the Assign button. We use the following mapping for the line-numbering macros. For *TurnOnLineNumbers*, we map the keys Ctrl+0, Ctrl+N. (The *N* stands for numbering.) We've mapped the *TurnOffLineNumbers* macro to Ctrl+0, Ctrl+Shift+N. Figure 3-8 shows how that shortcut looks after we've assigned it to the macro.

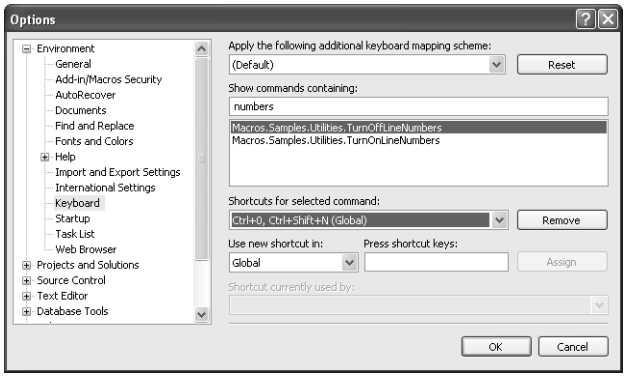


Figure 3-8 The *TurnOffLineNumbers* macro with a shortcut key assignment

Setting up line numbering is fairly straightforward, but will we want to toggle line numbering often enough to justify the brain cells it'll take to remember the aliases and the shortcuts we just created? Maybe. But if not, we can store these commands on a menu and then find them there when we need them. To add these macros to a menu, right-click on a toolbar in Visual Studio and choose Customize. On the Commands tab of the Customize dialog box, you'll find a Macros category. With Macros selected, scroll through the Commands list until you find Samples.Utilities.TurnOnLineNumbers. Choose that command and drag it to a menu. The menu you drag it to will expand, and you can place your selected command precisely where you want it.

To customize the new menu command, right-click on it to bring up a shortcut menu. (The Customize dialog box must remain open.) You can rename the new command Turn On Line Numbers to make it a little more readable. Do the same with the *TurnOffLineNumbers* macro, as shown in Figure 3-9.

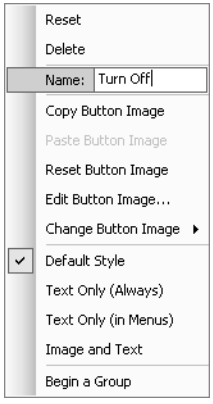


Figure 3-9 Adding the *TurnOffLineNumbers* macro to a menu

You can add a macro such as this one to a toolbar in just the same way. In that case, you'll probably want to specify a button image to use with the macro that you're adding.

Outlining

The Visual Studio outlining feature is probably familiar to programmers who've used other advanced editors. The idea is to group code by functionality to make it easier to navigate the code in the Code Editor. Figure 3-10 shows a code file in which the outline has been collapsed to the methods in the file. At the end of each collapsed line, you'll see a box with an ellipsis in it. Hover your mouse pointer over that box to display a ToolTip that shows some of what's in the collapsed node.

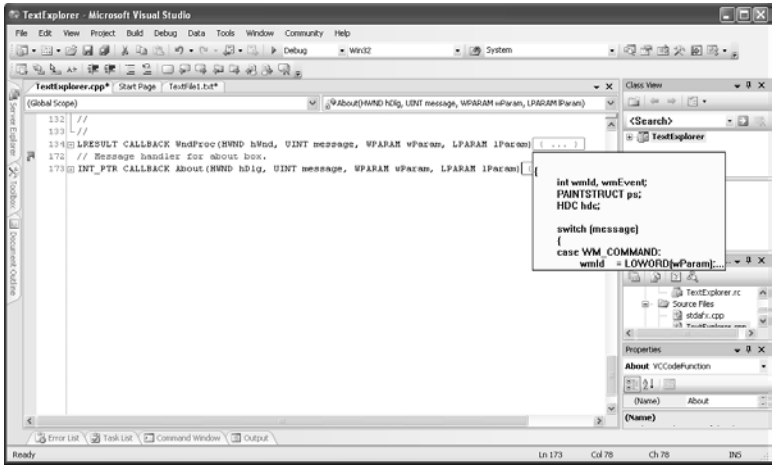


Figure 3-10 A ToolTip indicates the contents of a collapsed node when outlining is enabled.

You can turn off outlining by pressing Ctrl+M, Ctrl+P (*Edit.StopOutlining*). Turning off outlining makes the outlining indicators along the side of the Code Editor disappear completely. You can restart outlining by pressing Ctrl+M, Ctrl+O (*Edit.CollapsetoDefinitions*). The *CollapsetoDefinitions* command will restart outlining in the Code Editor and will collapse each code block in the file. Pressing Ctrl+M, Ctrl+L (*Edit.ToggleAllOutlining*) will open all the collapsed blocks in the Code Editor. You can also type **Edit.StartAutomaticOutlining** from the Command Window to restart outlining.

In addition to hiding logical code blocks in the Text Editor, you can collapse an arbitrary selection in Visual C++ by selecting some text and pressing Ctrl+M, Ctrl+H (*Edit.HideSelection*). This function can be very handy for collapsing some code between two distant points in a code file, and it even works to collapse lines in a plain-text file. If you want to expand the collapsed selection, press Ctrl+M, Ctrl+U (*Edit.StopHidingCurrent*).

Table 3-5 lists the shortcuts associated with outlining in the Text Editor.

Table 3-5 Outlining Shortcuts

Command	Keystroke	Named Command
Stop outlining	Ctrl+M, Ctrl+P	Edit.StopOutlining
Toggle outlining	Ctrl+M, Ctrl+L	Edit.ToggleAllOutlining
Toggle expansion	Ctrl+M, Ctrl+M	Edit.ToggleOutliningExpansion
Hide selection	Ctrl+M, Ctrl+H	Edit.HideSelection
Stop hiding selection	Ctrl+M, Ctrl+U	Edit.StopHidingCurrent

Programming Help

A number of features in the IDE make it easier for programmers to write code. You should be familiar with Help in the IDE, so we won't talk too much about it. Help in Visual Studio is very simple—you just select what you don't understand and press F1. You almost always get what you're looking for. In this section, we'll go over some of the features of the IDE that you've probably used but that you might not be so familiar with.

IntelliSense

IntelliSense is one of those features that you start to rely on heavily as a programmer. It's a timesaving feature that can really help you do the right thing when you're typing code into the IDE. IntelliSense provides statement completion in the form of context-sensitive member lists that appear automatically as you type code into the Code Editor. These lists can save you a great deal of time when you're programming an unfamiliar API, and they can help you reduce errors that would normally be caught only at build time.

IntelliSense works by parsing the code you type into the Code Editor based on the project type context. This means that your source file needs to be part of a project or a solution before IntelliSense kicks in. IntelliSense is mostly automatic. It works on code that's part of the .NET Framework, and it works on external methods from references you've added to your project. It even works on XML Web services references that have been added to a project.

IntelliSense provides four major types of functionality when you're working with a supported language. Most programmers will use these four features—statement completion, parameter information, word completion, and code comments—in their automatic form; that is, they'll take the information as presented in the IDE without thinking too much about what's being shown. That's an absolutely valid way to use the technology. If this approach works for you and doesn't get in your way, IntelliSense is doing exactly what it's designed to do. You can also employ IntelliSense more deliberately by using the shortcuts associated with displaying IntelliSense information.

You might want to turn statement completion and parameter information off if you find them distracting, in which case you'll need to use the shortcuts, named commands, or toolbar buttons associated with the various IntelliSense features to display this information. To turn off statement completion in the Code Editor, go to the Text Editor folder in the Options dialog box. Select the language you want to apply your changes to, or select All Languages if you want to apply your changes universally. Open the General page, and, in the Statement Completion section, clear the Auto List Members check box. You can also toggle the Parameter Information setting from this page.

Note You can increase the number of options returned in a member list by clearing Hide Advanced Members. Hide Advanced Members is the default setting for Visual Basic, so you might be missing a number of possible completions if you leave that setting cleared.

With statement completion turned on, IntelliSense presents you with information as soon as you type an operator as part of a statement. If you have statement completion turned off or if you want to display this information immediately, press Ctrl+J. The result is shown in Figure 3-11.

```
try
{
    RssFeed.
}
catch
{
    Messa                    a valid RSS feed.", "Not a valid RSS feed.",
    return;
}
```

Equals
FromText
FromUri
ReferenceEquals

Figure 3-11 Forcing statement completion by pressing Ctrl+J

To select an entry to complete a statement, use the up and down arrow keys to navigate to the desired completion and then press Tab.

To force parameter information like that shown in Figure 3-12, press Ctrl+Shift+Spacebar. This gives you a list of the parameter overloads you can choose from for a particular method. With the parameter information showing, use your up and down arrow keys to view the available parameters.

```
Console.Write(
```

▲17 of 18▼ void Console.Write (string format, object arg0, object arg1, object arg2)
format: The format string.

Figure 3-12 Viewing the parameter information for a method by pressing Ctrl+Shift+Spacebar

Use the parameter information provided by selecting the item that best suits your needs and then type the parameters into your method. You'll notice that after you type each parameter, the next parameter in the list appears in bold. Watching for this pattern helps ensure that every parameter in the list is entered correctly.

You might not be too familiar with the word completion feature if you're used to using IntelliSense automatically. Word completion lets you type in a few characters of a particular statement and get a list of possible completions for that statement. This functionality is a bit different from statement completion, which gives you a member list based on context. Word completion simply gives you a list of all the possible completions for the letters you've typed in. The shortcut for word completion is Alt+Right Arrow, but the statement completion shortcut (Ctrl+J) also works.

Finally, you might have noticed that when you hold your mouse pointer over an identifier in the Code Editor, ToolTip information appears. This ToolTip information is part of the Quick Info feature of IntelliSense, which contains the declaration for the identifier and any associated code comments. You can force this information by using Ctrl+K, Ctrl+I. You can add code comments to any method in Visual C# by typing /// (three forward slashes) on the line directly above the method definition. Even if you're not going to create documentation for your methods, IntelliSense makes code comments such as these helpful for letting another developer figure out how to use your code.

Brace Matching

Automatic brace matching is an IntelliSense feature that helps you determine whether braces in your code are matched properly. There are two types of brace matching in Visual Studio 2005. Rectangle brace matching happens when the cursor is next to the brace. Highlight brace matching works in Visual Basic, Visual C#, and Visual C++ and goes into effect when you type a closing brace into the Code Editor. The brace types affected include parentheses *()*, brackets *[]*, and braces *{}*. In addition, the conditional macro expressions *#if*, *#else*, and *#endif* are matched as you type the closing expression, and quotation marks are matched when you type the closing set.

Using the Command Window

If you've used a modal editor such as Vim (Vi improved) for a number of years and are used to typing editor commands at a command line, the Command Window in Visual Studio will come as a welcome surprise. I've already referred to using the Visual Studio Command Window a number of times (in both Chapters 1 and 2), but it's worth considering the various ways you can use this tool in your everyday work.

Tip You can clear the Command Window by entering the *cls* command.

As you've probably noticed by now, the named commands in Visual Studio generally map to menu commands in the IDE. So if you want to use a named command from the Command Window, all you usually need to do is to type the name of the menu containing the command and then the dot operator and the name of the command. For example, if

you want to search by using the Command Window, you first bring the window to the front by pressing Ctrl+Alt+A. To open the Find dialog box, type Edit.QuickFind in the Command Window. You'll notice that some commands, such as the *Edit.QuickFind* command, can take arguments. This means that you can search from the Command Window without having to deal with a dialog box. Whether or not you find this approach better is a matter of personal preference.

There are two ways to get to a command prompt in Visual Studio. The way that we've described in the book so far is to open the Command Window by using the Ctrl+Alt+A shortcut. You can also type commands into the Find combo box on the Standard toolbar by pressing Ctrl+D (*Edit.GoToFindCombo*). Normally, typing text in the Find combo box simply gives you a quick way to search the currently open document for a term or phrase. When you type a > (greater than) character into the box, the box changes to one capable of taking commands. You can then type named commands in the box that you would usually type in the Command Window. Because both the Command Window and the Find combo box support IntelliSense, you can simply type a named command to see all the possible completions for the command. You can see the list of completions in Figure 3-13.

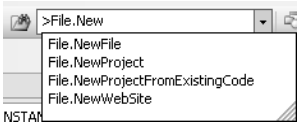


Figure 3-13 Command completion in the Find combo box

Search, Replace, and Regular Expressions

If you can't easily search for and replace text in your editor, you're probably not working with a very good editor. It's great to be able to type in code easily, but finding and fixing code problems are tasks you must do often as a programmer, and the search functions built into the editor are what make that work easy (or difficult). Visual Studio offers a number of ways to search in the Text Editor, and it offers a powerful regular expressions facility that allows you to perform extremely complex searches.

First let's take a look at the named commands and shortcuts associated with the Find and Replace operations in Visual Studio. These might be familiar to you because they're mapped to the Common User Accessibility (CUA) shortcuts that you might have used in Windows or in Office.

To bring up the Find and Replace dialog box, press Ctrl+F. You can click the plus button to expand the Find Options area in the dialog box, as shown in Figure 3-14. The options in this dialog box are fairly straightforward. You can specify case (Match Case), whole-word searches (Match Whole Word), and the direction of your search (Search Up). An interesting

option on this dialog box is Search Hidden Text. When this check box is left cleared, text that is hidden in a collapsed node of an outline won't be searched.

Figure 3-14 The Find and Replace dialog box in Find mode

The Find and Replace dialog box in Visual Studio is actually a tool window, so you can dock it in the IDE or even toggle the Dockable option on the shortcut menu available from the title bar to make it a tabbed window in the center of the IDE.

You can dock the Find and Replace dialog box by dragging it to a side of the IDE. But making it a floating dialog box might be preferable because it's easy to accidentally dock the window when you're trying to get it out of your way. You can turn off docking by choosing Floating on the shortcut menu for the dialog box. (Right-click on the Close button on a dialog box to open its shortcut menu.) Doing so will let you drag the Find dialog box around the screen with impunity.

You can bring up the Find and Replace dialog box in Quick Replace mode by pressing Ctrl+H. This dialog box contains a combo box for the Replace With text. Using the Find and Replace dialog box is straightforward. Especially handy are the Search options, which let you choose between the current document, all open documents, the current project, and specific selections or blocks in the editor.

Using the Find In Files option in the Find and Replace dialog box (Ctrl+Shift+F for Find or Ctrl+Shift+H for Replace), shown in Figure 3-15, make it fairly easy to find text within a project or a directory structure. Setting up these file searches takes a little more work than performing a standard Find command. You can type the path you want to search in the Look In field or you can click the ... button, which opens the Look In dialog box. You can use this dialog box to narrow your search to your project or to a directory structure on your machine. The output from the Find In Files search is sent to the Find Results window by default.

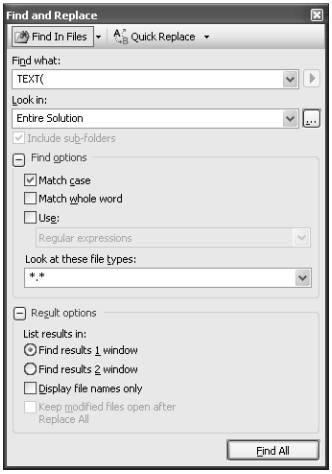


Figure 3-15 Find In Files from the Find and Replace dialog box

The Visual Studio Find and Replace shortcuts are listed in Table 3-6.

Table 3-6 Common Search Shortcuts

Command	Keystroke	Named Command
Find	Ctrl+F	Edit.QuickFind
Replace	Ctrl+H	Edit.QuickReplace
Find In Files	Ctrl+Shift+F	Edit.FindinFiles
Replace In Files	Ctrl+Shift+H	File.ReplaceinFiles

Incremental Searching

Incremental searching is a feature of Visual Studio that's a real timesaver. An incremental search is performed one character at a time, matching each word in the search string from the position of the cursor. You can start an incremental search by pressing Ctrl+I. You'll see the mouse pointer transform into a down arrow like the one shown in Figure 3-16.

```
1  // TextExplorer.cpp : Defines the entry point for the application.
2  //
3
4  #include "stdafx.h"
5  #include "TextExplorer.h"
6
7  #define MAX_LOADSTRING 100
8
9  // Global Variables:
10 HINSTANCE hInst;                              // current instance
11 TCHAR szTitle[MAX_LOADSTRING];                // The title bar text
12 TCHAR szWindowClass[MAX_LOADSTRING];          // the main window class name
13
14 // Forward declarations of functions included in this code module:
```

Figure 3-16 Starting an incremental search by pressing Ctrl+I

With the down arrow showing, start typing the term you want to search for. As you type, words that match the letters are matched starting from the top. When you've completed the pattern you want to search for, press Ctrl+I again to move to the next match. You can continue to press Ctrl+I until you reach the end of the document to match every instance of the term you're searching for. If you want to search upward, just press Ctrl+Shift+I. It works in the same way. You can exit the incremental search by pressing Enter or Esc.

Looking Ahead

This chapter has given you a pretty good idea of how named commands apply to the Code Editor in the IDE and how you can customize the IDE by using alias commands in the Command Window, through keyboard shortcuts, and by adding menu items associated with named commands and macros. In Chapter 4, we'll tell you about some of the community features in the IDE.

Chapter 4

Community Content and VSTemplates

In this chapter:

Microsoft® Visual Studio® was designed to be an upgradeable and evolving program. In this chapter, we'll show you how to use some of the new features of Visual Studio to make your programming more powerful by sharing code with other users.

Community Content

In the early hobbyist days of programming, people would often gather in small groups to share ideas about how to program or to share code. These user groups would often have a shared goal in a program that they needed to write. Maybe they wanted to write a program to balance their checkbooks, keep track of recipes, or maybe share software algorithms that would make software development easier and faster. This community of software developers still exists today, but because of tools such as the Internet, the community has grown to a global scale. Today, sites such as Microsoft's own GotDotNet.com, WindowsForms.com, and Asp.net, have sprung up, letting a user in Pittsburgh collaborate on a software project with a developer in Tokyo. You can also use the tools built into Visual Studio on the Community | Search menu to open the Microsoft Developer Network (MSDN®) help browser, where you can search for content.

Visual Studio makes collaboration between developers easier with tools designed specifically to allow you to share code and ideas. If you need help, simply go to one of your favorite programming community Web sites and make a request. Another user might have experience with the problem you are trying to solve and have some code that is similar to what you need. He or she can package the code and distribute it, either by sending the content to you directly through e-mail or by posting it to a Web site. Sharing code is the focus of the Community Content Installer—a tool that takes code created by another user and places it on your computer so that you can start using it right away.

Note The Content Installer also gives you an opportunity if you are in the business of selling components for Visual Studio. You could package controls for the Toolbox, code snippets, and other items, and then sell those components from your Web site. Customers can then use the Content Installer to install the components they purchased.

Installing Content

Content that you install on your computer using the Content Installer is contained in a file with the extension .vsi. This file is a compressed file containing all the files necessary to correctly install data onto your computer. The data that can be installed includes controls that you can drag and drop from the Toolbox onto a WinForm or a Web form; code snippets that make writing blocks of code as simple as pressing a few keys on the keyboard; templates for creating new projects and project items; or add-ins and macros, which are small programs that customize the Visual Studio integrated development environment (IDE). When you receive content from another user, installing that content for use is as easy as working through a wizard and selecting which items to install. After you receive a package of content enclosed in a .vsi file, you can open the file by either double-clicking the file if it is stored on disk, or, if you are downloading the file directly from the Internet, you can click the Open button on the download dialog box from your Web browser. Figure 4-1 shows the first page of the Content Installer showing a .vsi file that contains a control for the Toolbox.

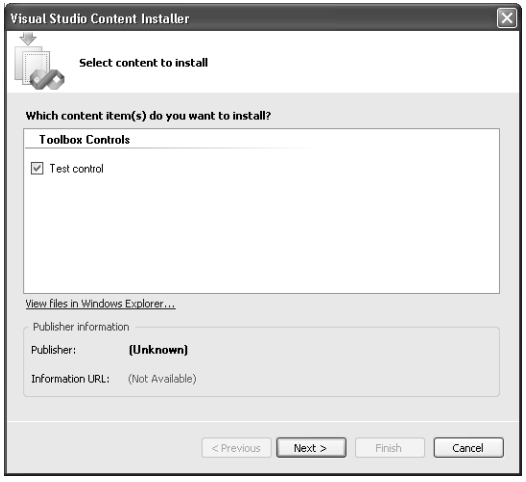


Figure 4-1 The Content Installer with a content file open and ready to install

Controls do not have any configuration information to display to the user, so clicking the Next button will take you directly to the final page of the wizard. Some content elements, such as code snippets, do allow you to configure how they are to be installed, so the wizard includes a configuration page for them. After you have configured the installation, if applicable, you are ready to begin installing the content. The page indicating that content is ready to install is shown in Figure 4-2.

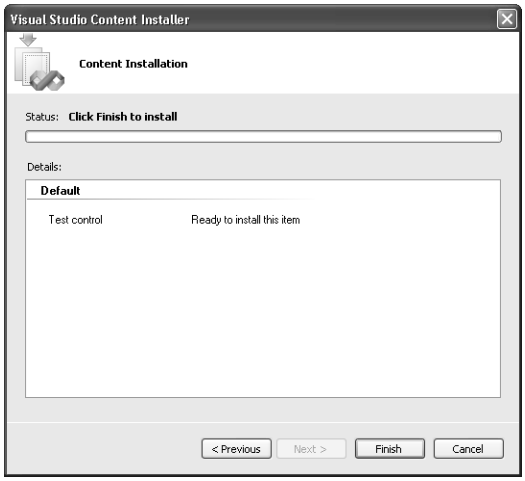


Figure 4-2 Ready-to-install content

Click the Finish button to install the files. During installation, there might be a conflict between a file to be installed and an existing file of the same name. If that happens, you are given the chance to overwrite the file, skip installing the file, or rename the file to a suggested file name. When installation is completed successfully or with errors, you will be presented with a dialog box like the one in Figure 4-3.

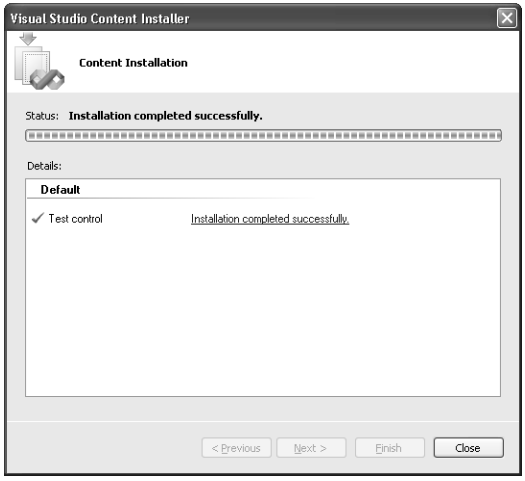


Figure 4-3 The content Installer after items have been installed

After installing an item, a hyperlink will appear next to the item that was successfully or unsuccessfully installed. You can click this link to see the Install status report, which consists of a list of files installed for that content item as well as any status message returned. If installation of any item failed, you can use the status information to correct the problems, and then return to the wizard to click the Finish button again to try to reinstall the unsuccessfully installed items.

Security

As with any file that you place on your computer, you should first make sure that you trust the files and know what you are installing; items installed by means of the Content Installer are no different. The first page of the wizard has a hyperlink labeled "View files in Windows Explorer...." Clicking this hyperlink will open a Microsoft Windows® Explorer window displaying the files that are within the .vsi file, allowing you to view the files and inspect the contents of those files to verify that they will do no harm. Also on the first page of the Content Installer is an area that displays information about the .vsi file. If the .vsi file to install was signed with an Authenticode® signature, the name of the company that created the content, as well as a link to more information about that content, will appear. If the file has not been signed, then this information will not be available.

Note If you click the Review button to view the files to be installed, you should be careful not to delete any of the files from within the Windows Explorer window. If you do delete any files, installing the content might fail.

Creating Downloadable Content

You can also easily create your own content to share with others. A .vsi file contains one or more files that are compressed using the .zip file format with the extension renamed from .zip to .vsi. In addition to the installable content files, the .vsi file contains a manifest file with the .vscontent extension. The .vscontent file uses the XML format and specifies not only which files to install, but also how they are to be installed. When the Content Installer opens the .vsi file, it first decompresses the file and searches for the first available file with the .vscontent extension. If such a file is present, it is opened, the XML within it is read, and then the Content Installer wizard appears, ready to start installing.

The VSContent File Format

The .vscontent file schema was designed to be very simple and easy to create. The XML file for content starts with the tag <VSContent>. This tag has one attribute that determines the XSD schema used not only for ensuring that the XML is correct, but also by the Visual Studio XML editor to give statement completion information when the file is opened for editing. The contents of the most basic .vscontent file are as follows:

```
<VSContent xmlns="http://schemas.microsoft.com/developer/vscontent/2005">
</VSContent>
```

Next we need to specify the items that will be installed. Each content item is enclosed within a tag named <Content> and lists the name and description of the item to display to the user with the <DisplayName> and <Description> tags, respectively. These tags contain any text that you want to display to the user identifying the content in the first page of the Content

Installer—but keep it short because there is not much space available for display. To identify the type of content to the Content Installer, the <FileContentType> and <ContentVersion> tags are used. <FileContentType> can contain one of the following values: Toolbox Control, Macro Project, Addin, VSTemplate, or Code Snippet. <ContentVersion> specifies the version of the content to be installed and is generally the string "1.0." The last required tag (or tags, because you can specify more than one of them) is the <FileName> tag. This tag lists the files to install for the content item. The file names listed in <FileName> tags are relative to the location of the .vscontent file. If a file to install (let's suppose the file name is File.xyz) is zipped from the same directory as the .vscontent file, the file name is File.xyz; if the file is stored in a subdirectory of the folder containing the .vscontent file, the file name is *SubDirectoryName*\File.xyz.

When all of these tags are put together with a list of the files that are to be installed, you have a .vscontent file that can be loaded by the Content Installer to install on your computer or somebody else's. An example of a .vscontent file, which can be used to install controls onto the Toolbox, is given here:

```
<VSContent xmlns="http://schemas.microsoft.com/developer/vscontent/2005">
 <Content>
  <FileName>CompanyName\Control.dll</FileName>
  <DisplayName>Test control</DisplayName>
  <Description>A control to test with</Description>
  <FileContentType>Toolbox Control</FileContentType>
  <ContentVersion>1.0</ContentVersion>
 </Content>
</VSContent>
```

Figure 4-1 showed a screenshot of the Content Installer with this .vscontent file open. Here, the Control.dll file is stored in a folder named CompanyName, whereas the .vscontent file is assumed to be in the same folder containing the folder CompanyName. You can combine multiple Content items within a .vscontent file, allowing you to distribute multiple items at one time or to group items that are related and dependent on one another.

Installing Templates and Starter Kits

Creating new projects or new files that go within projects is one of the most common, and yet more complicated, tasks that a developer may perform. If you needed to create a file with a simple class within that file, the task isn't overly complex—you simply add a new blank file to the project and add the class. However, imagine having to create a Windows form from scratch without a starting file generated by a wizard, and you can see how hard it would be to create the WinForm. Templates and starter kits are packages of files that you can use to quickly generate a new file or project. We will discuss how to generate these packages later in this chapter, but for now, let us examine the Content Installer XML necessary to install a template or starter kit.

A .vscontent file used to install a template file looks very much like the sample .vscontent file that we saw earlier. The <FileContentType> name for a template installer is VSTemplate. This

content type installs files only with the .zip extension. Three different attributes are required to specify a content item for a template. The .vscontent XML schema allows attributes to be attached to a content item to provide custom data to the installer, directing the installer on how the item should be installed. Attributes use a name and value pair not only to name the data, but also to contain the data. The three attributes that a VSTemplate content item requires are *TemplateType*, *ProjectType*, and *ProjectSubType*. The value for a *TemplateType* attribute can be one of *Project* or *ProjectItem*, and it declares where the template can be used. If this value is *Project*, the template is copied into a location so that you can create the project through the New Project dialog box. If this value is *ProjectItem*, the file is copied into a location so that you can right-click in the Solution Explorer on a project or a folder within a project, choose New Item from the Add menu, and choose the template to add to an existing project. The second attribute necessary within a .vscontent file for a VSTemplate is the *ProjectType* attribute. This attribute specifies the programming language used for the template. Suppose you have a Microsoft Visual C#® project open; you would not want your Microsoft Visual Basic® template to appear within the Add New Item dialog box. And if you have the New Project dialog box open with the Visual Basic project type selected, you do not want your Microsoft Visual J#® project to appear. With the *ProjectType* attribute, you can create a filter that determines in which dialog box the template will appear. The possible values for this attribute are *Visual C#*, *Visual Basic*, *Visual J#*, and *Web*. (Although Web is not a programming language, it is a type of project.) The final attribute is the *ProjectSubType* attribute, which gives either the platform of the template if *ProjectType* is Visual C#, Visual J#, or Visual Basic; or the programming language of the project if *ProjectType* is Web. If the *ProjectType* is Web, the possible values for *ProjectSubType* are Visual C#, Visual Basic, or Visual J#. If the *ProjectType* is not Web, possible values for *ProjectSubType* are Windows, Smart Device, Database, or starter kits.

When the Content Installer copies the template file to disk it examines the values of these three attributes and uses them to construct a destination path where the template is placed. The installer begins with the Visual Studio user data directory, which is C:\Documents and Settings*username*\My Documents\Visual Studio 2005, and then the installer appends Templates to this path because this is the storage location for all installed templates. Next the installer looks at the *TemplateType* attribute. If it is *Project*, it adds *ProjectTemplates* to the path, or, if the attribute value is *ProjectItem*, it adds *ItemTemplates* to the path. Next the value of *ProjectType* is added, and finally the *ProjectSubType* attribute value is added.

The following .vscontent file defines a Visual C# console application template named MyConsoleApplication.zip for Windows. Based on the values given in the XML, the .zip file for the template is installed into the location C:\Documents and Settings*username*\My Documents\Visual Studio 2005\Templates\ProjectTemplates\Visual C#\Windows.

```
<VSContent xmlns="http://schemas.microsoft.com/developer/vscontent/2005">
  <Content>
    <FileName>MyConsoleApplication.zip</FileName>
    <DisplayName>My Console Application</DisplayName>
    <FileContentType>VSTemplate</FileContentType>
```

```
    <ContentVersion>1.0</ContentVersion>
    <Attributes>
      <Attribute name="ProjectType" value="Visual C#"/>
      <Attribute name="ProjectSubType" value="Windows"/>
      <Attribute name="TemplateType" value="Project"/>
    </Attributes>

 </Content>
</VSContent>
```

When the Content Installer has installed this .vscontent file, you will be able to create a console application within the new project dialog box. The template will appear under the node Visual C#\Windows in the tree on the left side of the dialog box and in the My Templates section of the list on the right side of the dialog box.

Wizards, Templates, and Starter Kits

When you create new projects or items for a project, you are likely to see the terms wizards, templates, and starter kits. Each is a different method for creating new code.

The most basic technique for creating code is through a wizard. A wizard is a COM object that you write from scratch to perform all the work necessary to generate a project or a project item. The template wizard, which consumes templates to generate projects or project items, is an implementation of a wizard. You do not need to write code—you need only to provide the sources for a project or project item and the template wizard handles re-creating the project or project item for you.

A starter kit is a template. A template .zip file and a starter kit .zip file physically are identical—the only difference is in the terminology. A starter kit is a project template that generally builds a more complex project. Visual Studio ships with a couple of starter kits, such as one you could use for a screen saver application or a Movie Collection tracking program. These projects obviously perform specific tasks, whereas a WinForm application simply serves as a starting point for your application.

Installing Controls to the Toolbox

Controls are DLLs that are inserted into the Toolbox, which the user can drag and drop onto a WinForm or ASPX Web page designer to quickly create the user interface for an application. These pre-built components save users a lot of time by allowing them to quickly design a program without needing to write lots of code. You can use the Content Installer to install controls in the Toolbox. To install a control, a DLL containing the control must be copied into a place where it can be found, and then Visual Studio must be started so that it can load and then add the control to the Toolbox.

Placing the control into a place where Visual Studio can find it is easy—at least with the Content Installer. When the Content Installer installs a control for the Toolbox, the control

is placed into the directory My Documents\Visual Studio 2005\Controls*CompanyName*, where *CompanyName* is the name of your company, group, or any other unique name that you want to use. When the Content Installer installs controls, it will use this *CompanyName* to create a new tab in the Toolbox (if one with that name does not already exist), and then the controls contained within the DLLs in this directory will be added to that Toolbox tab. This procedure allows you to create a tab in the Toolbox that distinguishes your controls from the default ones installed by Visual Studio or other companies.

Note You can also place controls into the directory My Documents\Visual Studio 2005\Controls. If you place a DLL here, the controls the DLL contain will be placed on a tab named My Controls.

The second step to install the control is to invoke Visual Studio, Visual Basic Express, Visual C# Express, and so on to allow those programs to install the control. Installing controls can be costly in terms of startup performance, so the control installer will start each of these applications to place the controls on the Toolbox with the command *programname /Command Tools.InstallCommunityControls* where *programname* is the full path to the executable for the various editions of Visual Studio. The command *Tools.InstallCommunityControls* invokes code within Visual Studio to search for, and then install the controls in the Toolbox. (We will more fully discuss commands in Chapter 7.) You can also use the command *Tools.InstallCommunityControls* outside of the control installer to reinitialize any controls you might have installed but that do not appear in the Toolbox. Suppose you install Visual C# Express, and later you decide to upgrade to Visual Studio; do you need to reinstall all the controls you might have downloaded? No. Simply start Visual Studio, open the Command Window tool window, type **Tools.InstallCommunityControls**, and then press Enter. Visual Studio will look for any controls that have previously been installed with the Content Installer and place them on the Toolbox.

When we were discussing the .vscontent file format, we saw a sample .vscontent file containing the information necessary to install a control. The name of the content type for a control is Toolbox Control, which should be placed within the <FileContentType> tag. In this example, a tab named CompanyName will be created within the Toolbox, and any controls in the file Control.dll will be added to that tab.

```
<VSContent xmlns="http://schemas.microsoft.com/developer/vscontent/2005">
 <Content>
  <FileName>CompanyName\Control.dll</FileName>
  <DisplayName>Test control</DisplayName>
  <Description>A control to test with</Description>
  <FileContentType>Toolbox Control</FileContentType>
  <ContentVersion>1.0</ContentVersion>
 </Content>
</VSContent>
```

This also means that you need to create a folder named CompanyName in the folder containing the .vscontent file, and you would place the Control.dll file within the CompanyName folder.

Installing Code Snippets

Code Snippets are a source code editor productivity enhancer. If you are editing a C# file and you need to create, for example, a *for* loop, you could manually enter the text for that *for* loop, or you could use a snippet to create the loop for you. To use a snippet, first type the word **for**; this will cause the IntelliSense® window to appear. If you were to press the Tab key twice with the *for* keyword selected in the IntelliSense window, Visual Studio will automatically generate the basic structure of a *for* loop, as shown in Figure 4-4.

```
for (int i = 0; i < length; i++)
{

}
```

Figure 4-4 The basic structure of a *for* loop, as generated from a snippet

The control variable is named *i*, but you might not always want this variable to have that name. To change this variable name, just type in the new name. When you are finished typing, press the Tab key, and the variable name *i*, as well as all uses of the variable *i* within the snippet code, will be renamed, as shown in Figure 4-5, where the variable name is changed to *j*.

```
for (int j = 0; j < length; j++)
{

}
```

Figure 4-5 The basic *for* loop, with the control variable renamed to *j*

The variable name length is highlighted next, and it, too, can be edited. After typing the number **10** (creating a loop over the numbers 0 to 9, inclusive), the *for* loop appears like the one in Figure 4-6.

```
for (int j = 0; j < 10; j++)
{

}
```

Figure 4-6 The basic *for* loop, with the upper limit set to 10

Although this is not a lot of text to type, imagine typing this bit of code over and over, just as you do during your development work, and you can see the amount of keystrokes you will save. Snippets are stored in an XML format and can be installed in the same manner as other content items.

A .snippet file can contain many different snippets, but each file can contain snippets for only one particular programming language. The Visual Basic, Visual C #, Visual J #, and

XML editors currently support snippets. When installing a snippet, the <Content> tag must contain an <Attribute> tag indicating which programming language the snippet supports. The name of this attribute is *lang*, and the possible values are *vb*, *csharp*, *jsharp*, and *xml*. The following .vscontent file contains two items to install: a Visual Basic and a Visual C# snippet:

```
<VSContent xmlns="http://schemas.microsoft.com/developer/vscontent/2005">
 <Content>
  <FileName>VBSnippets.Snippet</FileName>
  <DisplayName>Visual Basic snippets</DisplayName>
  <Description>Visual Basic snippets to install.</Description>
  <FileContentType>Snippet</FileContentType>
  <ContentVersion>1.0</ContentVersion>
  <Attributes>
   <Attribute name="lang" value="vb"></Attribute>
  </Attributes>
 </Content>
 <Content>
  <FileName>CSSnippet.Snippet</FileName>
  <DisplayName>C# snippets</DisplayName>
  <Description>C# snippets to install. </Description>
  <FileContentType>Snippet</FileContentType>
  <ContentVersion>1.0</ContentVersion>
  <Attributes>
   <Attribute name="lang" value="csharp"></Attribute>
  </Attributes>
 </Content>
</VSContent>
```

Installing Add-ins and Macros

In Chapters 5 and 6, we'll see how to create macros and add-ins, but while we are discussing installing files, we can show you how to install these types of files. An add-in (at least an add-in written using a .NET Framework programming language) consists of at least one file, and possibly more. The required file is a file with the .addin extension. This file describes the add-in to Visual Studio and provides information such as the name and description of the add-in. The second possible file is a .dll that contains the code for the add-in. The content type for an add-in is Addin, and this is one of the few content types that will accept more than one <FileName> tag to specify more than one file to install. This content type does not need any <Attribute> tags. Here is a simple .vscontent file used to install an add-in:

```
<VSContent xmlns="http://schemas.microsoft.com/developer/vscontent/2005">
 <Content>
  <FileName>MyAddin.Addin</FileName>
  <FileName>MyAddin.dll</FileName>
  <DisplayName>An add-in to install.</DisplayName>
  <Description>This is a test add-in.</Description>
  <FileContentType>Addin</FileContentType>
  <ContentVersion>1.0</ContentVersion>
 </Content>
</VSContent>
```

A macro, much like an add-in, is a file that contains program code that the user can run to customize and modify data within Visual Studio. The difference between a macro and an add-in is that the user can edit and modify the macro code, and there is an editing and debugging environment devoted to just this type of code. The XML used to install a macro has the <FileContentType> of Macro Project, and the only file types that can be installed by the macro installer are files that end with a .vsmacros extension:

```
<VSContent xmlns="http://schemas.microsoft.com/developer/vscontent/2005">
 <Content>
  <FileName>Samples.vsmacros</FileName>
  <DisplayName>A macro to install.</DisplayName>
  <Description>A macro project to install.</Description>
  <FileContentType>Macro Project</FileContentType>
  <ContentVersion>1.0</ContentVersion>
 </Content>
</VSContent>
```

Add-in and macro content installers are a bit different than the other installer types in that these content types will install only for Visual Studio. The Express versions of Visual Studio do not support macros or add-ins, so although templates, snippets, and controls can be installed for all those versions of Visual Studio, add-ins and macros will be available only from within the Enterprise, Pro, or other premium versions of Visual Studio.

Zipping

After you have created the content you want to distribute, the next step is to package the file for distribution. You do this by zipping up the files—you can use either the zip functionality built into Windows XP or any other zipping tool to do this—then rename the file to have the extension .vsi rather than .zip. If you are using Windows Explorer to zip the files, simply select the files to zip, right-click each file, and from the shortcut menu, choose Send To and then Compressed (zipped) Folder from the submenu. If you are using another zipping tool, such as WinZip, you need to make sure that you do not use some of the more advanced options these zip utilities provide. For example, you shouldn't use encryption or a nonstandard compression algorithm, and you shouldn't save files with full path information to zip your files. If you do, then extraction could fail within the Content Installer when the file is opened because the zip utility that the Content Installer uses supports only basic zipping functionality.

Note The extension .vsi has its roots in the file extension .msi, which stands for Microsoft installer. VSI is short for Visual Studio installer. The two technologies, however, should not be confused with one another. The Content Installer cannot be used to install .msi files, and the Microsoft installer cannot be used to install .vsi files.

After you have renamed the .zip file to have the extension .vsi, you can either e-mail or post the file to a Web site so that friends or colleagues can install the content that you have

created. But before you make the .vsi file available to others for use, you will want to test the file to make sure it works. You could just double-click the .vsi file from within Windows Explorer, but if you find a bug in your content, you would need to correct the problem, then re-zip and rename the extension, and repeat. Rather than taking the time to go through the zip-rename process, you can just double-click the .vscontent file. Both the .vsi and .vscontent extensions are associated with the Content Installer. You can also use the Content Installer to browse to a .vsi, .zip, or .vscontent file. From the Run dialog box in the Windows Start menu, navigate to C:\Program Files\Common Files\Microsoft Shared\MSEnv and run the command vscontentinstaller.exe /browse. This will display the standard Windows Open File dialog box, where you can browse to any file you need to open and install the content the file contains.

Signing Your Content

Some people might be wary of installing content on their computers from unknown sources, but if content has been identified as coming from a trusted source, they might not be as concerned about installing that content. You can purchase Authenticode certificates from certificate authorities such as Thawte.com or VeriSign.com. But there is a problem: you cannot sign .zip files (or .zip files renamed to have the .vsi extension)—you can sign only .dll, .exe, or .cab files. To enable you to sign your .vsi files, Visual Studio has a utility named MakeZipExe, which will take a .zip file and create a self-extracting .exe that you can then sign by using the SignCode tool, a utility that is part of the .NET Frameworks software development kit (SDK). To create a .vsi file that you can sign, first generate your .zip file by using your favorite zip utility just as you would for unsigned content, but do not rename the file extension. Then, from the command line, run the following command:

```
MakeZipExe –zipfile:"path to your .zip file"
```

This command will create an .exe file in the same location as the zip with the same file name except for the extension. Next you will need an Authenticode certificate and the signcode.exe utility. Because the command line arguments to the signcode.exe tool can vary between the different certificate authorities, you should search the Web site of your certificate issuer for assistance in using the Signcode.exe tool. After you have signed the .exe file, you can rename the file to have the .vsi extension. When the Content Installer opens the .vsi file, it will examine the file to determine if the file is an unsigned .zip file or a signed .exe file and extract the content appropriately.

When signing code, you should keep two things in mind. First, certificates are not cheap. Certificates can cost from a few hundred dollars to thousands of dollars depending on the certificate, so you will need to decide if signing your code is worth the cost. If your target audience is consumers buying controls from your Web site, you probably will want to sign your .vsi. If you are only sharing your files with friends, you probably will not need to sign. The second thing about signing is that the Content Installer knows how to read self-extracting .exe files generated by using only the MakeZipExe tool; other tools, such as WinZip, do not produce .exe files compatible with the Content Installer.

The ContentBuilder Utility

Creating .vsi files is not overly complicated, but it can be made easier. The samples for this book include the sources to a tool named ContentBuilder. This tool provides a graphical user interface (GUI) for building a .vsi file just by selecting files on your computer's disk drive.

Implementing Your Own Downloadable Types

As we have seen, the Content Installer offers different content types that you can install: templates, add-ins, macros, code snippets, and controls. However, you might have an idea for another kind of content that could help users better collaborate with one another. The Content Installer is extensible, allowing you to create your own content-type installers. In this section, we'll show you how you can extend the Content Installer with your own installer types.

Creating the Project

A custom installer is simply a .NET class library DLL that the Content Installer will load and run when a .vscontent file has a <FileContentType> tag value that corresponds to the name you give to your own custom installer. The easiest way to get started creating your content installer is to use a starter kit. Accompanying the samples for this book are two starter kits named *Content Installer UI-Less Page* and *Content Installer UI Page*. Both appear in the Visual C# section of the New Project dialog box. The first of these starter kits is used to create a project that does not have any configuration options in the Content Installer wizard. The second starter kit is used to create a project that displays user interface (UI) for configuration. When you create a project using the starter kit, the name of the project as entered in the New Project dialog box will become both the name of the installer and the text that you use within the <FileContentType> tag in a .vscontent file. The following sections describe how the code generated by these starter kits work, and how you can write code to interact with the Content Installer.

Interface Implementation

An installer is simply a class library that implements the interface *IImportCommunityContent*. This and other interfaces that you will use to program the Content Installer are defined in the assembly Microsoft.VisualStudio.VSContentInstaller.dll. The *IImportCommunityContent* interface has the following signature:

```
public interface IImportCommunityContent
{
    bool AddContentItem(IContentItem[] contentItems, IContentInstallerSite site);
    string Import(IContentItem contentItem);
    bool SupportsImportUI { get; }
    IImportPageData[] GetImportPages();
    void UpdateContentItemInstallStatus(IContentItem[] contentItems);
}
```

When the Content Installer loads your custom installer, the first method called is the *IImportCommunityContent.AddContentItem* method. When this method is called, all the data necessary for your installer to know which files and how they are to be installed is contained within the *IContentItem* array. One *IContentItem* object is passed to the *AddContentItem* method for each <Content> tag in the .vscontent file for the <FileContentType> that matches the type your installer installs. If you were to open a .vscontent file with two Addin content types and one VSTemplate content type, and if you were implementing the add-in installer, you would be handed two *IContentItem* objects, one for each Addin content item in the .vscontent file. *IContentItem* has a set of methods and properties for getting to data for the content in the .vscontent file. The properties *ContentVersion*, *Description*, *DisplayName*, and *FileContentType* on the *IContentItem* interface each map to the tag of the same name in the .vscontent file. The property *IContentItem.AttributePairs* returns an object of type *System.Collections.Specialized.StringDictionary* with one element for each Attribute tag within the .vscontent file. Your installer specifies any attributes that the installer requires, as well as the names and values of those attributes. *IContentItem* also exposes two methods, *GetFileNames* and *GetRootFileNames*. When the Content Installer reads the <Content> section of a .vscontent file, it gathers all the <FileName> tags together and stores them for later use. When you call the *GetRootFileNames* method, the list of file names as given in the .vscontent file is returned. *GetFileNames* also returns the list of file names, except that these file names are prepended with the directory in which the files are placed and can be used as the source of a copy operation. Assuming that you have a variable named *destinationPath* containing the path in which you are copying items, you can use code such as this to copy the files into the correct location:

```
foreach(VSContentInstaller.ContentItem
  contentItem in contentItems)
{
  string []sourceFileNames =
    contentItem.GetFileNames();
  string []rootFileNames =
    contentItem.GetRootFileNames();
  for (int i = 0; i < sourceFileNames.Length ; i++)
  {
    string combinedDirectory = Path.Combine(
      destinationPath, rootFileNames);
    Directory.CreateDirectory(
      Path.GetDirectoryName(combinedDirectory));
    File.Copy(sourceFileNames[i], combinedDirectory);
  }
}
```

The reason for these two methods is quite simple. Suppose the path given in a <FileName> tag contains a directory, MySubDirectory\File.ext, for example. If you were given only the source directory to copy the file from, you would not know how to re-create the directory MySubDirectory in the destination location. *GetRootFileNames* returns the path exactly as specified in the XML, and you can use this to recreate the necessary destination path.

The second parameter is an *IContentInstallerSite* object, which is an object implemented by the Content Installer that you call into to copy files, set status, and perform other operations. We will see the methods on this interface shortly.

After calling your *AddContentItem* method, the Content Installer will then call the *SupportsImportUI* property. Not all installers need to display a user interface to the user. In fact, the only installer that does show UI among the installers that ship with Visual Studio is the snippet installer. If your installer does need to display UI, then you should return false from this property, and if you do need to display UI, then return true. If you do return true, the Content Installer will call the *GetImportPages* method next. This method creates an array of class objects implementing the *IImportPageData* interface containing one element for each page your installer needs to display in the UI; it then returns this array. The *IImportPageData* interface has two different values you can set. *HeadlineText* is the text displayed in the banner at the top of the wizard when your UI page is active. *Page* is set to an instance of a class that derives from the *UserControl* class, and it is to be displayed when your page becomes active when the user is traversing the steps in the wizard. The *UserControl* that you create for your UI should have a size of 470 × 305 and can contain any UI elements that you want.

One thing to keep in mind is that there is not a 1:1 correspondence between the number of *IContentItem* objects passed to the *AddContentItem* method and the number of elements returned from the *GetImportPages* method. For example, a .vscontent file may contain 10 different snippet content items to install, so 10 different *IContentItem* objects are passed to the *AddContentItem* method. The code snippet installer takes these *IContentItem* objects, sorts them based on the programming language the snippet is written in, and then displays one page for each of the languages supported. If the .vscontent file contains 10 C# snippets, one *IImportPageData* object is returned. If the .vscontent file contains one C# snippet, one XML snippet, and eight Visual Basic snippets, three *IImportPageData* objects are returned.

After calling these methods and properties, the Content Installer is ready to show the first page of the content installer UI, such as that displayed in Figure 4-1 (on page 56). The first page displays a list of items, one for each *IContentItem* object, both handled by your installer and by others, that are to be installed. There is also a check box next to each content item, all selected by default. Your installer's *UpdateContentItemInstallStatus* method is called as the user selects and clears items in the Content Installer UI. This method call provides you with an array of *IContentItem* objects. Only content items that are to be installed will be passed, so if the user decides not to install one or more content items, you can add or remove them from the list of items to display within your user interface.

After the user has reached the last page of the Content Installer wizard, the Next button changes to the Finish button. When the user clicks the Finish button, the Content Installer starts informing each installer that it should install content with a call to the *Import* method. For each item that was selected on the first page of the Content Installer, the installer's *Import* method is called with the *IContentItem* that is to be installed. This is where you call

the code that we saw earlier to copy the files into the correct location. An error may occur in any setup operation. If any exception is unhandled by your installer, or if you throw an exception indicating that you detected an error, the Content Installer will catch that exception and indicate to the user that an error occurred. If an error did not occur, your *Import* method needs to return a string indicating that installation was successful.

The Site Interface

IContentInstallerSite is an interface exposed by the Content Installer to provide you a way to control the UI of the Content Installer and make development of your installer easier. The first method of this interface is the *CopyFile* method. Earlier we showed you some code to copy files to disk by using the *System.IO.File.Copy* method. But if you use the *CopyFile* method, the Content Installer will store a list of the files being installed for a content item, and the Content Installer will use this list for display within the Install status window. In addition, the *CopyFile* method handles problems such as when the file to copy already exists on disk. *CopyFile* accepts four parameters: the source file path, the destination file path, a value of type *DuplicateFileCase*, and an out parameter into which the path the file was placed is copied. The *DuplicateFileCase* parameter allows you to control how the file is copied if the destination file exists. If this value is anything other than *DuplicateFileCase.None*, and the destination file exists, a dialog box is given to the user allowing him to overwrite the existing file, to skip copying the file to disk, or to use a new file name suggested by the Content Installer. The values within the *DuplicateFileCase* enumeration will enable or disable the corresponding UI options within this dialog box.

When calling *CopyFile*, you should specify only the *EnableRename* or *EnableAll* enumerated values to the *allowRenameOfFile* if the file name is not referenced by another file. For example, if you are copying the files of a project, and the name SomeFile.cs exists on disk, and you are also copying a project file (SomeProject.csproj) that references the file SomeFile.cs, ensure that you do not pass *EnableRename* or *EnableAll* to *allowRenameOfFile*. Otherwise, when the user opens the project file, the file SomeFile.cs will try to load the original SomeFile.cs, not the renamed version.

The method *IContentInstallerSite.EnableNextButton* takes a Boolean value that allows you to enable or disable the Next button within the wizard. If you have UI displayed for your installer and the user enters data that is invalid, which should prevent him or her from navigating to the next page, you can call this method specifying false to prevent the user from going to the next page. After the user enters correct data, you can call *EnableNextButton(true)* to enable the button and to allow the user to continue.

GetApplicationData allows you to retrieve information specific to the application into which you are installing the content item. This method takes two values and returns an array of interfaces containing data that you can use to install your program. The first parameter is the content type that you are trying to install; the second is the version of the content type (both of which are stored in the .vscontent file). Upon return from *GetApplicationData*, an array

of *IApplicationHostData* contains one element for each application (such as Visual Studio, Visual Basic Express, Visual J# Express, and so on) that the content can be installed for. The *IApplicationHostData* interface has six properties that contain information from the registry, and these properties are *RegistryRoot*, *ApplicationName*, *UserDataFolder*, *ApplicationPath*, *ProgId*, and *ApplicationImage*. Each of these properties is valid for every edition of Visual Studio except the *ProgId* property. Visual Studio is the only version that supports an automation object model, so for any of the Express versions of Visual Studio, this property will return an empty string. Table 4-1 lists each of these properties and example values returned from these properties.

Table 4-1 *IApplicationData* properties

***IApplicationData* property**	**Example Value**	**Property Use**
RegistryRoot	*Software\Microsoft\VisualStudio\8.0*	This is the location where data for the application is stored in the registry. Prepend the registry hive key, either *HKEY_LOCAL_MACHINE* or *HKEY_CURRENT_USER*, as appropriate.
ApplicationName	*Microsoft Visual Studio 2005*	Display text for the name of the application.
UserDataFolder	*%USERPROFILE%\My Documents\Visual Studio 2005*	Location where user files are stored for the application.
ApplicationPath	*"c:\Program Files\Microsoft Visual Studio 8\Common7\IDE\devenv.exe"*	The path to the application.
ProgId	*VisualStudio.DTE.8.0*	COM ProgID for the application. This is not valid for the Express versions of Visual Studio.
ExpressVersion	*False*	Returns true if the program is an express version (such as Visual Basic Express), or false if the program is Visual Studio.
ApplicationImage		An image for the application. If the content item does not supply an image, the default Visual Studio logo (the "Infinity" icon) is used.

The last important method on the *IContentInstallerSite* is the *ShouldContinue* method. The Content Installer is multithreaded, allowing the user to cancel installation even while an installer is installing data to the computer. Periodically while installing, such as just before you call *CopyFile*, your installer should call this method to determine if it should continue. If at any time *ShouldContinue* returns false, you should immediately return from your implementation of *Import*.

Registration

After you have implemented the code for your installer, the final step is to let the Content Installer know how to find your code with a set of registry keys and values. All registry information for the Content Installer is under the key *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSEnvCommunityContent\ContentTypes*. Underneath this key, you create a key with the name of your installer, and you create two values named *Assembly* and *ClassName*. Assembly gives either the path or the strong name of the assembly implementing your installer, and *ClassName* is the full name, including namespace and class name, of the class implementing *IImportCommunityContent*. Underneath that is a key with the name *ContentHosts* and then a key with the version number of your installer, usually named 1.0. Next is a list of keys with the name of the editions of Visual Studio, which can be Visual Studio 2005, Visual Basic Express 2005, Microsoft Visual Web Developer™ Express 2005, Visual J# Express 2005, Visual C# Express 2005, and Microsoft Visual C++® Express 2005. If your installer does not support one of these editions of Visual Studio, you can omit that key. For example, if you are installing a set of C++ header files, you should generate keys only for Visual Studio 2005 and Visual C++ Express 2005. When you call *IContentInstallerSite.GetApplicationData*, one *IApplicationHostData* entry is returned for each version of Visual Studio that is registered. Underneath each of these keys is a set of values, each having the name as given in the left column of Table 4-1, and a value similar to that as in the center column of Table 4-1. Following along with this description of keys and values is probably not easy, so an example .reg file with all the registry values necessary to define an installer named MyContentType is given in Listing 4-1. This registry script creates all the necessary entries for Visual Studio and the Express versions of Visual Studio. If you were to use the starter kits to create a content installer, all this data would be pre-populated. All you would need to do is merge the .reg file in the system registry by double-clicking the file in Windows Explorer, and Registry Editor will create all the necessary registry settings for you. Notice that there are some values that contain text, such as %USERPROFILE%. When the Content Installer reads these values from the system registry, it will expand all environment variables into their set values.

Listing 4-1 An example registry script to register a content installer

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSEnvCommunityContent\
  ContentTypes\MyContentType]
"ClassName"="MyContentType.MyContentType"
"Assembly"="C:\Documents and Settings\CRAIGS\My Documents\
  Visual Studio\Projects\MyContentType\MyContentType\
  bin\debug\MyContentType.dll"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSEnvCommunityContent\
  ContentTypes\MyContentType\ContentHosts]

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSEnvCommunityContent\
```

```
  ContentTypes\MyContentType\ContentHosts\1.0]

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSEnvCommunityContent\
  ContentTypes\MyContentType\ContentHosts\1.0\Visual Studio 2005]
"ApplicationName"="Microsoft Visual Studio 2005"
"ApplicationPath"="C:\\Program Files\\Microsoft Visual Studio 8\\
  Common7\\IDE\\devenv.exe"
"RegistryRoot"="Software\\Microsoft\\VisualStudio\\8.0"
"UserDataFolder"="%USERPROFILE%\\My Documents\\Visual Studio 2005"
"ProgId"="VisualStudio.DTE.8.0"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSEnvCommunityContent\
  ContentTypes\MyContentType\ContentHosts\1.0\Visual Basic Express 2005]
"ApplicationName"="Microsoft Visual Basic Express 2005"
"ApplicationPath"="C:\\Program Files\\Microsoft Visual Studio 8\\Common7\\IDE\\
vbexpress.exe"
"RegistryRoot"="Software\\Microsoft\\VBExpress\\8.0"
"UserDataFolder"="%USERPROFILE%\\My Documents\\Visual Studio 2005"
"ExpressVersion"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSEnvCommunityContent\
  ContentTypes\MyContentType\ContentHosts\1.0\
  Visual C# Express 2005]
"ApplicationName"="Microsoft Visual C# Express 2005"
"ApplicationPath"="C:\\Program Files\\Microsoft Visual Studio 8\\
  Common7\\IDE\\csexpress.exe"
"RegistryRoot"="Software\\Microsoft\\VCSExpress\\8.0"
"UserDataFolder"="%USERPROFILE%\\My Documents\\Visual Studio 2005"

"ExpressVersion"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSEnvCommunityContent\ContentTypes\
MyContentType\ContentHosts\1.0\Visual C++ Express 2005]
"ApplicationName"="Microsoft Visual C++ Express 2005"
"ApplicationPath"="C:\\Program Files\\Microsoft Visual Studio 8\\Common7\\IDE\\
vcexpress.exe"
"RegistryRoot"="Software\\Microsoft\\VCExpress\\8.0"
"UserDataFolder"="%USERPROFILE%\\My Documents\\Visual Studio 2005"

"ExpressVersion"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSEnvCommunityContent\
  ContentTypes\MyContentType\ContentHosts\1.0\
  Visual J# Express 2005]
"ApplicationName"="Microsoft Visual J# Express 2005"
"ApplicationPath"="C:\\Program Files\\Microsoft Visual Studio 8\\
  Common7\\IDE\\vjsexpress.exe"
"RegistryRoot"="Software\\Microsoft\\VJSExpress\\8.0"
"UserDataFolder"="%USERPROFILE%\\My Documents\\Visual Studio 2005"

"ExpressVersion"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSEnvCommunityContent\
  ContentTypes\MyContentType\ContentHosts\1.0\
  Visual Web Developer Express 2005]
```

```
"ApplicationName"="Microsoft Visual Web Developer Express 2005"
"ApplicationPath"="C:\\Program Files\\Microsoft Visual Studio 8\\Common7\\IDE\\
vwdexpress.exe"
"RegistryRoot"="Software\\Microsoft\\VWDExpress\\8.0"
"UserDataFolder"="%USERPROFILE%\\My Documents\\Visual Studio 2005"

"ExpressVersion"=dword:00000001
```

An Example—Samples Installer

Now that you know how to implement an installer, you can try out a very simple installer, the samples installer. Samples are used extensively by programmers for tips on how to more effectively use an API. Available in this book's companion content, this samples installer allows you to package samples and redistribute them to another user through a .vsi file. The samples installer is so simple that you already saw the majority of the code when we described the *IContentInstallerSite.CopyFile* method. The only additional code that it uses is some security checks to ensure that the files are installed in places that will not harm the user's computer.

Security Attributes

Unless they are written with security in mind, installers could be used to place files onto disk that probably should not be installed. As an extra layer of protection, there are two attributes that you can place on your content installer class to restrict which content is passed to it. This minimizes the impact of rogue content that a user could download and blocks installers that spoof existing installers. The first of these attributes is the *ContentInstallerContentTypeRestrictionAttribute* attribute, which takes as a parameter the name of a content type. When the Content Installer loads an installer, the Content Installer will look for this attribute and, if found, will compare the content type to be installed with the value passed to this attribute. If they match, the content will be permitted to be installed; otherwise, the content will not be allowed to install, and a security exception will be generated. The string passed to *ContentInstallerContentTypeRestrictionAttribute* should match the name that you give to your installer within the system registry.

The second attribute that you can place on your content installer class is *ContentInstallerSupportedFileSecurityAttribute*. This attribute allows you to filter out the types of files that your installer can install. The installer for macro projects should be allowed to install only macro projects (files with the extension .vsmacros), not executable files (files with the extension .exe), so the installer for macros uses this attribute declaration: *ContentInstallerSupportedFileSecurity(".vsmacros")*. There is the possibility that a macro could reference an external DLL, but to minimize the possibility of bad code being installed, the macro project installer allows only .vsmacros files to be installed. You should be equally security-minded when creating your installer. Allow installation of only the minimum list of file types, not everything that could possibly be installed. If your installer needs to install multiple file types, you can specify the *ContentInstallerSupportedFileSecurityAttribute* attribute

multiple times. This is a portion of the class used to install add-in file types, and it specifies that it can install .addin and .dll files:

```
[ContentInstallerContentTypeRestriction("Addin")]
[ContentInstallerSupportedFileSecurity(".addin")]
[ContentInstallerSupportedFileSecurity(".dll")]
class AddinInstallerPage : IImportCommunityContent
{
    ...
}
```

Your installer also needs to make sure that the files that are being installed are installing to the location you intended. Suppose you wanted to install your content into the folder My Documents\Visual Studio 2005\MyContentType; the installer places .exe files on disk, and the .vscontent file gives a destination path of \Windows\notepad.exe. When your installer runs this .vscontent file, it will try to overwrite the Notepad program with code that could be harmful to the computer. The next time the user tries to run the Notepad program, that malicious code will run. Therefore, you should check where you are installing content before you call *IContentInstallerSite.CopyFile*. The *CopyFile* method does not check this path for you before copying the file because there may be content installers that have a valid reason to install content into a path such as the Windows directory.

Creating VSTemplates

Probably the most common type of content that you will be sharing with others is templates. In earlier versions of Visual Studio, if you wanted to create a project or project item that the user could add to the New Project or Add New Item dialog boxes for cloning, you had to write a wizard. Writing a wizard to generate even a simple project was not easy, so to make this process easier, Visual Studio 2005 offers a new wizard technology that makes creating reusable projects and project items easier than ever. This technology, VSTemplates, enables you to quickly create a new project or project item to import into a solution or project by putting together an XML file that describes the files that make up your project or project item. When the user runs your template through the New Project or Add New Item dialog boxes, Visual Studio will start the template wizard, which then reads and processes the XML file, creating the project or project item for you. The VSTemplate wizard turns generating projects and project items into a data-driven process. Each item that appears in the New Project and Add New Item dialog boxes, when added to a solution or an existing project, uses the VSTemplate wizard to generate the project and project items. And of course, you can create your own template files to appear within these dialog boxes.

Using the Export Template Wizard

The easiest way to get started creating a VSTemplate is to use the Export Template Wizard. This wizard will examine a project or project item that is loaded into a solution or project, and it will generate a VSTemplate that you can use within the Add New Item or New Project

dialog boxes to create new project items or projects. You can also package these files into a .vsi file for easy distribution to other users. This wizard is available from the File menu, and it will quickly generate a .zip file containing a .vstemplate file and the files necessary to re-create an item within a project or a project and its contents.

To use the wizard, you must first have one or more Visual Basic, Visual C#, Visual J#, or Web projects opened in a solution. After you start the wizard, you will see a dialog box like that shown in Figure 4-7.

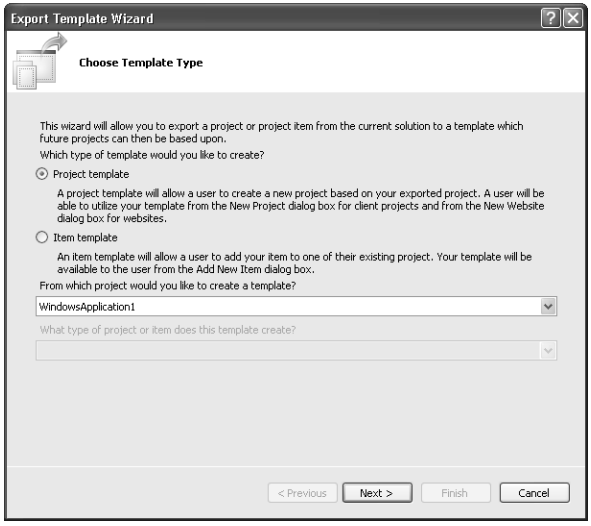


Figure 4-7 The first step of the Export Template Wizard

This dialog box gives you two different options. The first, Project Template, will generate a template for a project. The Item Template option, if selected, will enable you to export one single file from within a project. After you choose to export either a project or project item, you then select which project to export, or the project containing the item to export. This is done from within the first drop-down box, which gives a list of all projects that are within the currently open solution. The final drop-down box on this page is only visible when you have selected a Web project to export and allows you to select the programming language for the Web project.

If you select to export a project item and then click the Next button, the dialog box shown in Figure 4-8 will appear.

Here, you can navigate through the tree to find an item to turn into a template. If you were to select one item and then select another, the check box on the first item will be cleared. Some items, such as forms or controls, can contain multiple items. For example, a Windows form could have a Form1.cs file, a Form1.designer.cs file, and a Form1.resx file. The dependent items (Form1.designer.cs and Form1.resx) will not appear in this step of the wizard, but if you select the Form1.cs file, even though you do not see those items in the tree, they will

automatically be exported. When you click the Next button, a list of assemblies referenced by the project in which the item is located is given. (See Figure 4-9.) When the template is added to a project, any references that you select in this step of the wizard will also be added to the project. Suppose you are adding a Windows Form to a console application. If you were to select the check box next to the *System.Windows.Forms* assembly, the user importing your form will not need to manually add the reference to *System.Windows.Forms* because the template wizard will take care of adding that reference for you.

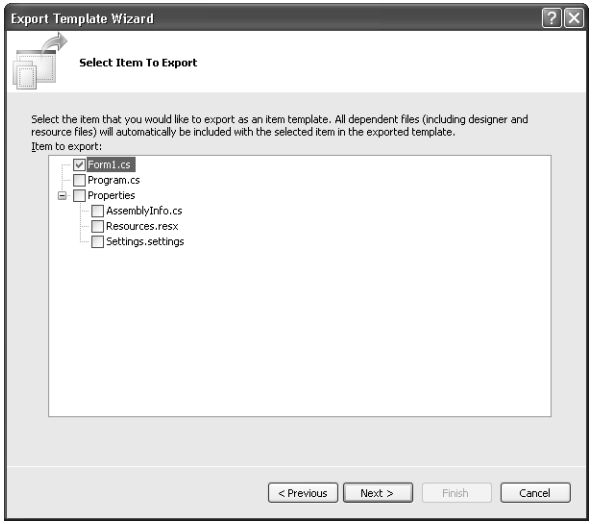


Figure 4-8 Selecting a project item to export

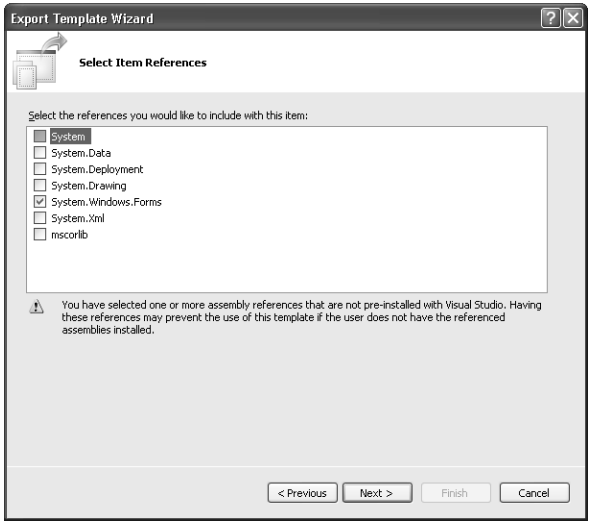


Figure 4-9 Selecting the references to import when the project item is added to a project

The final page of exporting a project item is the same as the final step of exporting a project. (See Figure 4-10.) This step in the wizard allows you to select an icon to use within the New Project or Add New Item dialog boxes. It also allows you to specify the text to show beneath the icon by changing the *Template name* field. The description of the template shown within the New Project and Add New Item dialog boxes can be changed by modifying the *Template description* field.

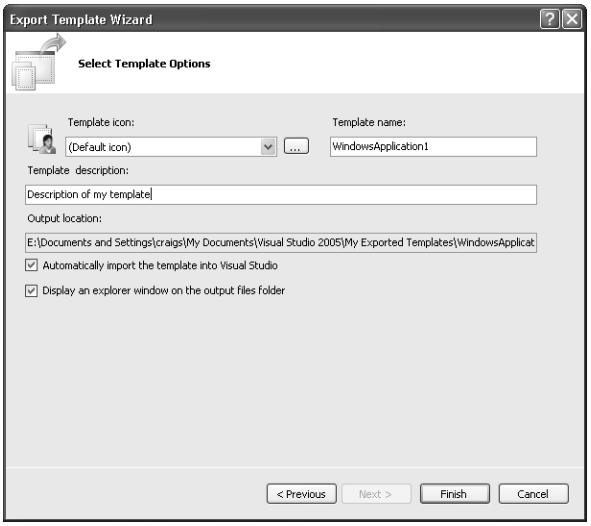


Figure 4-10 The final page of the Export Template Wizard

After you click Finish, the wizard will gather the data that you entered, generate a .vstemplate file, package together all the files that make up your project, and then create a .zip file containing this data. If the Automatically Import The Template Into Visual Studio check box is selected, the .zip file is placed into the correct location so that if you were to open the New Project or Add New Item dialog boxes immediately after exporting the template, the template appears in the My Templates section of these dialog boxes. A copy of the template is also stored into the C:\Documents and Settings*username*\My Documents\Visual Studio 2005\My Exported Templates folder, making it easier for you to find the template and send it to others, or to package it into a .vsi file.

Creating Templates by Hand

Although the template wizard will quickly generate a project or project item template for you, there could be times when you want to customize the template beyond what the Export Template Wizard automatically generates. To create a template, you need a set of files that will produce the sources for the new project or new project item that the user will create. The template wizard supports only Visual Basic, Visual C#, Visual J#, and Web projects, so your base project must be of one of these types. After you have created your base project or project item, you then need to go through those files and insert special strings,

called *replacement tokens*, into the source and project files. The Export Template Wizard automatically scans all the file s in the project for specific strings, such as the name of the project, and replaces those strings with a replacement token. When the template wizard processes your source files, the files are searched for the replacement tokens. When one is found, the token is replaced with a value that is dependent on the state of your computer and the files that are currently open inside of Visual Studio. For example, suppose your company used a common header at the top of each source file that looked like the following:

```
//------------------------------------------------------------------
//  Class1.cs
//
//  (C) Copyright 1999 My Company.
//
//  Contents:   My source code file
//
//  Owner:      UserName
//
//  Revisions:  02/07/2005 14:24:35    Created by UserName
//
//------------------------------------------------------------------
```

You could modify your template file so that when processed by the wizard, it is automatically modified to contain the relevant information. The modified template would look like this:

```
//------------------------------------------------------------------
//  $itemname$
//
//  (C) Copyright $year$ $registeredorganization$.
//
//  Contents:   My source code file
//
//  Owner:      $username$
//
//  Revisions:  $time$    Created by $username$
//
//------------------------------------------------------------------
```

There are many different replacement values available for use within your project files. Table 4-2 lists these values, as well as possible values and a description.

Table 4-2 Replacement Values

Replacement Variable	Example Value	Description
$guid1$, $guid2$, $guid3$, $guid4$, $guid5$, $guid6$, $guid7$, $guid8$, $guid9$, $guid10$	e3593046-f53f-48f9-9c6e-c1761196384f	These variables are used for generating globally unique identifiers (GUIDs) within source code. They can be used when a unique value is necessary, including for generating COM object code.
$time$	02/07/2005 14:24:35	The time that the source code file was generated.
$year$	2005	The year that the source code was generated.

Table 4-2 Replacement Values

Replacement Variable	Example Value	Description
$username$	Craigs	The name of the user who is currently logged on to the computer.
$userdomain$	Redmond	The name of the domain, if available, that the computer is a member of. At Microsoft, the name of the domain that most users belong to is Redmond.
$machinename$	CraigsLaptop	The name of the computer on the network.
$clrversion$	v2. 0.50215	The version of the .NET Framework that is being used by Visual Studio at the time the file is processed to build the program.
$registeredorganization$	Microsoft	The name of the organization that owns the license for the operating system. This value is entered when installing the operating system when the user is prompted for the user name and organization.
$itemname$	My ClassFile	The file name as entered by the user, but with the extension removed.
$safeitemname$	My_ClassFile	The file name as entered by the user, but modified so that the n ame can be used as an identifier. Any character that would not be recognized as a valid identifier character is replaced with an underscore. This name does not include the extension.
$itemrootname$	My Form1.vb	This is the name of an item being added to either a new or existing project.
$safeitemrootname$	My_Form1.vb	The same as $itemrootname$, but in a form that can be used as a programmatic identifier.
$fileinputname$	Form1	Available only to Add New item templates, this is the file name entered into the Add New Item dialog box. If the name entered into the dialog box is Form1.MyForm.vb, this value will be Form1. MyForm.
$fileinputextension$	.vb	Available only to Add New Item templates, this is the extension of the file name entered into the Add New Item dialog box.
$rootnamespace$	WindowsApplication1	Available only to Add New item templates, this is the default namespace, as specified in the project properties window, for the project. If an item is added to a folder of a project, this value will also include the folder name. For example, adding a file to a folder named NewFolder will cause $rootnamespace$ to be WindowsApplication. NewFolder.

Table 4-2 Replacement Values

Replacement Variable	Example Value	Description
$runsilent$	true/false	This value is set to true when any user interface that the wizard may display should be hidden, and it's set to false if UI can be shown
$wizarddata$		Within the VSContent XML file, you can create a tag named <WizardData> under the document node. Any data within this tag is passed along through this value.
$rootname$	Form1.vb	This is the complete file name entered in the Add New Item dialog box.
$projectname$	My Project1	The name of the project.
$safeprojectname$	My_Project1	The name of the project, but modified in a way that allows you to use the name as an identifier in your source code.
$installpath$	C:\Program Files\ Microsoft Visual Studio 8\Common7\IDE	The directory in which Visual Studio is installed.
$exclusiveproject$	true/false	If this value is false, the project is being added to an existing project; otherwise, the project is being added to a new solution.
$destinationdirectory$	C:\Documents and Settings\craigs\My Documents\Visual Studio 2005\Projects\ MyProjectName	The directory into which the new project is being created.

Note Not only can you create your own templates by hand, but you can also modify templates generated with the Export Template wizard. Simply export a template, open the .zip file containing the template, and then modify the files to your liking. When you are done, add the modified files to the .zip file.

The VSTemplate Schema

After you have modified your project or project item to contain the replacement tokens, you now need to create an XML file that describes to the template wizard how the project or project item should be re-created. This file, which has the extension .vstemplate, begins with a <VSTemplate> tag and has three XML attributes. The *Version* attribute specifies the version of the wizard that the XML file is designed to work with. For Visual Studio 2005, the version number is 2.0.0. The second attribute, *Type*, specifies the type of items that can be generated with the .vscontent file. Currently, three types of items are supported: *Project*, *Item*, and *ProjectGroup*. If you are creating

a new project template, the value is *Project*; if you are creating a new project item, the value is *Item*. The template wizard also supports creating multiple projects at a time within the solution. Suppose you need to create a Web Service application and a console application that consumes that Web Service. With a template type of *ProjectGroup*, you could create both of these projects at once rather than create them separately. Finally, the *xmlns* attribute lists the XML Schema of the file. The following XML is the most basic of .vstemplate files:

```
<VSTemplate Version="2.0.0" Type="Project" xmlns=
  "http://schemas.microsoft.com/developer/vstemplate/2005">
</VSTemplate>
```

A .vstemplate file has two main sections. The *TemplateData* section describes the visual representation of the template in the UI. The second section, *TemplateContent*, details each file that is part of your template and how those files should be re-created in the target solution or project.

The *TemplateData* Section

The *TemplateData* section is within the VSTemplate section of the .vscontent file. This section determines the appearance of templates within the New Project and Add New Item dialog boxes. An example of this section looks like this XML fragment, which defines a custom C# class library template:

```
<TemplateData>
  <Name>My Class Library</Name>
  <Description>A project for creating a C# class library (.dll)</Description>
  <Icon>AnIcon.ico</Icon>
  <ProjectType>CSharp</ProjectType>
  <SortOrder>20</SortOrder>
  <DefaultName>ClassLibrary</DefaultName>
  <ProvideDefaultName>true</ProvideDefaultName>
</TemplateData>
```

The tags of this XML have the following meanings:

- ***Name*** This is the name of the item shown underneath the icon within the New Project or Add New Item dialog boxes.
- ***Description*** This is the text shown when the user selects the icon in the New Project or Add New Item dialog boxes, giving the user more information about what type of project will be created when the template is processed.
- ***Icon*** A path, relative to where the .vstemplate file is located, to the icon to display within the New Project or Add New Item dialog boxes.
- ***ProjectType*** Can either be CSharp, JSharp, VisualBasic, or Web. This controls which node of the tree on the left side of the New Project dialog box the item will appear under, or, if the template is an item template, it specifies the project that the item can be added to.

- ***SortOrder*** This is the priority of the item within the New Project or Add New Item dialog boxes. The lower this value is, the higher it will appear to the top of these dialog boxes.
- ***DefaultName*** This is the default name of the new project or project item that appears in the New Project or Add New Item dialog boxes. Visual Studio takes this name and appends a value onto the end of the name starting at 1. If a project or project item with that name exists, the number is incremented until a unique name is found.
- ***ProvideDefaultName*** If this is true, then the Name field of the New Project or Add New Item dialog box will contain a default value based upon the DefaultName value. Otherwise, the project name is <Enter name>, and the user must manually enter a valid project name to continue.

The *TemplateContent* Section

The *TemplateContent* section is where you specify the directory structure layout on disk for your project. When the template wizard opens your .vstemplate file, it reads this section and copies the listed file or files into a temporary location. From this temporary location, the files are added to the solution (if the .vstemplate specifies a new project or project group) or added to the project (if the .vstemplate specifies a project item). This section looks different depending on whether you are creating a project, a project item, or a group project. An example *TemplateContent* section for creating new projects, taken from the C# class library template, looks like this:

```
<TemplateContent>
  <Project File="ClassLibrary.csproj" ReplaceParameters="true">
    <ProjectItem ReplaceParameters="true"
      TargetFileName=
      "Properties\AssemblyInfo.cs">
      AssemblyInfo.cs</ProjectItem>
    <ProjectItem ReplaceParameters="true"
    OpenInEditor="true">
      Class1.cs</ProjectItem>
  </Project>
</TemplateContent>
```

Here, a project file with the name ClassLibrary.csproj is expected to be in the same folder as the .vstemplate file. This project file, along with two files, AssemblyInfo.cs and Class1.cs, are copied into the temporary folder. Class1.cs will be copied into the same folder as ClassLibrary.csproj, but for AssemblyInfo.cs, two file names are given, AssemblyInfo.cs and Properties\AssemblyInfo.cs. If only one file name is given (because the *TargetFileName* attribute is not supplied), the source file is read from the same folder containing the .vstemplate file, and the file is copied into the project destination folder. Because, in this example, two file names are given in the <ProjectItem> tag, a folder named Properties will be created and the AssemblyInfo.cs file will be copied into that folder This allows you to fine-tune exactly how the directory structure for a project or project items are re-created on disk. You could also give a relative path for

the text of the ProjectItem tag, such as Properties\AssemblyInfo.cs. When the template wizard processes that item, the AssemblyInfo.cs file is copied from a subfolder named Properties into the folder named Properties within the temporary folder. Thus, you can write the line of XML to copy the AssemblyInfo.cs into a Properties folder using these three styles:

```
<ProjectItem ReplaceParameters="true"
  TargetFileName="Properties\AssemblyInfo.cs">
  AssemblyInfo.cs
</ProjectItem>
<ProjectItem ReplaceParameters="true">
  Properties\AssemblyInfo.cs
</ProjectItem>
<ProjectItem ReplaceParameters="true"
  TargetFileName="Properties\AssemblyInfo.cs">
  Properties\AssemblyInfo.cs
</ProjectItem>
```

The first line requires the Assemblyinfo.cs file to be in the same folder as the .vstemplate file, whereas for the second and third possibilities, the template wizard expects the file to be in a folder named Properties. All three styles will copy the file into a folder named Properties.

Note It is important to distinguish a project file from the *TemplateContent* section of a .vscontent file for creating projects. A project file, such as a file ending in the extensions .csproj, .vbproj, or vjsproj, defines the programming language–specific layout of a project on disk and how that project is loaded into Visual Studio. A .vstemplate file is a programming language–agnostic way of defining a project or project item. It might seem unnecessary to have both a project and a template file because they both contain similar data, but a separate .vstemplate file has its purpose. A group project template and project items do not have an associated project file, so the .vstemplate file is needed in that case. In addition, because the .vstemplate file is independent of language, you can use one file format to create projects for multiple languages, even languages not created by Microsoft.

In all three files from the example .vstemplate file, the *ReplaceParameters* attribute is set to true, meaning that files will be opened in the temporary folder and examined for replacement parameters, and if any are found, they will be replaced. If you do not have any replacements to make in a file, you can set this value to false (or not even give the attribute—it defaults to false) and increase performance by a little bit, because if this value is false, the steps to open the file and scan for replacement tokens that will not exist can be bypassed.

This sample also makes use of the *OpenInEditor* attribute. If this attribute is set to true, when the wizard finishes building the project or project item, it will open all the files with this attribute. With the *OpenInEditor* attribute, you can also optionally use the *OpenOrder* attribute. This attribute takes an integer value and lets you control the order in which documents are opened. If you were to use the *OpenInEditor*="*true*" attribute on multiple files,

then all the files will be opened, but only one will be the active window in the Visual Studio UI when the wizard is complete. If you use the *OpenOrder* attribute, the file with the highest value is opened last and will then be active in the UI.

An example <TemplateContent> section for creating new project items, which was taken from the C# class template, looks like this:

```
<TemplateContent>
  <References>
    <Reference>
      <Assembly>
      System, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089
      F</Assembly>
    </Reference>
  </References>
  <ProjectItem ReplaceParameters="true">Class.cs</ProjectItem>
</TemplateContent>
```

A new project item template is similar to new project templates, except that a project item template does not have an associated project file, so the <Project> tag is removed and the <ProjectItem> tag becomes a child of the <TemplateContent> tag. Also, a <References>, <Reference>, and <Assembly> tag has been added to the XML. After the Class.cs file has been added to the project, the template wizard will use the XML XPath query for References/Reference/Assembly tags, and, if found, the wizard will add the assembly named to the list of references for the project the item is being added to. Here, the System assembly is referenced by its full, strong name. The <References> tag is available only for new item templates and not in new project templates. Any references for project templates are already specified within the project file, so references for new projects are not necessary. You can use the <References> tag to prepare a project so that it will build correctly without the user needing to manually add references.

The final type of template is the project group. If you are trying to package a group project for installation in a VSI file, then the TemplateType attribute should be set to ProjectGroup. Rather than using the Project tag as you do for new projects, the <ProjectCollection> tag is used within the <TemplateContent> tag. Within this tag, you can combine the <SolutionFolder> tag to create a new solution folder within the solution and a <ProjectTemplateLink> tag to reference another .vstemplate file. If the <ProjectTemplateLink> tag is given within a <SolutionFolder> tag, the new project will be created within the new solution folder. Any <ProjectTemplateLink> or <SolutionFolder> tags that appear directly under the <ProjectCollection> tag will create the project or solution folder in the location where the project group is added to the solution.

```
<TemplateContent>
 <ProjectCollection>
  <SolutionFolder Name="Folder1">
   <ProjectTemplateLink ProjectName="ConsoleApp1">
```

```
      ConsoleApplication\csconsoleapplication.vstemplate
    </ProjectTemplateLink>
   </SolutionFolder>
   <ProjectTemplateLink ProjectName="ConsoleApp2">
      ConsoleApplication\csconsoleapplication.vstemplate
   </ProjectTemplateLink>
  </ProjectCollection>
 </TemplateContent>
```

<SolutionFolder> has one attribute, *Name*, that is used to name the solution folder that is created, whereas <ProjectTemplateLink> has one optional attribute, *ProjectName*. The value of this attribute is used to name the new project, but if it is not given, the name of the .vstemplate file without the extension, in both uses of the <ProjectTemplateLink> tags in this example, is *csconsoleapplication*. When a <ProjectTemplateLink> tag is encountered, the template wizard will gather together all the necessary information (such as where the project is to be created, the name of the new project, and so on), and then spawn off a new instance of the template wizard and create the new project. All replacements in the subproject are processed just as if the project were being created as a new project, and any values from the subproject (such as replacement values) will not propagate back up into the project group wizard replacements. When the new projects are added to the solution, the .vstemplate of those subprojects are searched for relative to the location of the group project .vstemplate file. In this example, the folder containing the group project template is prepended to ConsoleApplication\csconsoleapplication.vstemplate, and then the template wizard is run on the template file located at this computed path.

Wizard Data

The <WizardData> section of the .vstemplate file allows you to store freeform data in the .vstemplate file. One or more of these XML blocks can appear as a child of the <VSTemplate> tag, and one such <WizardData> XML fragment is shown here:

```
<WizardData Name="MyWizardData">Some user defined data here.</WizardData>
```

The data within a <WizardData> tag can be of any XML-representable data you may want, meaning it can be more XML, or, as in this example, just some plain text. When the template wizard reads in the .vstemplate file, any VSTemplate/WizardData tags that are found are read into memory, the *Name* attribute is read, a $ character is added to the beginning and end of the name, a replacement value is created with this name, and then the replacement value is set to the data within the <WizardData> tag. So, for this example, a replacement variable named *$MyWizardData$* is created and the value of this replacement is set to *"Some user defined data here."* You can then use this replacement value in your source files. This will allow you to create custom replacement values and make replacements in your files just by modifying the .vstemplate file. You could even pack all the contents of source files into the .vstemplate file and then create files that have nothing more than the replacement argument name.

Storing the Template on Disk

After you have created your .vstemplate and source files, you need to place them in a location so that Visual Studio can find and display your templates in the New Project and Add New Item dialog boxes. But first, you must place all the necessary files into a .zip file—simply zip all the files, such as the .vstemplate, source, and project files, into one .zip file. You do not need to rename the file to have a special extension as you do for .vsi files; the extension .zip will do. After the files have been zipped, you need to copy the file into one of four folders so that they can be found. We have already seen the path of two of these folders when discussing the VSTemplate content installer. The folder C:\Documents and Settings*UserName*\My Documents\Visual Studio 2005\Templates has two subfolders, ItemTemplates and ProjectTemplates. When the New Project or Add New Item dialog boxes are shown, Visual Studio will examine the appropriate directory for new, deleted, or modified .zip files and make the necessary updates to show the correct template in the dialog box. However, only the user whose My Documents folder has been modified will see changes made to these two folders.

What if you need to install a template that is available for all users of a computer? To make a VSTemplate available to all users, you need to place the template file in a location that all users can read from. The My Documents folder is readable only by the owner of that folder, so this is not an appropriate location for templates all users can invoke. The folders that are available to all users and where Visual Studio will look are either C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\ProjectTemplates for project or group projects, or C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\ItemTemplates for project items. Under these folders are subfolders such as CSharp, JSharp, VisualBasic, and Web. Within each of these folders are more folders, which further qualify how and in which dialog box the template can be invoked by the user. After you have selected the folder in which to place your template, you then need to force Visual Studio to recognize the template. Because the .zip files for a VSTemplate are stored in the Program Files folder, a location that only users with elevated permissions (such as an Administrator) can write to, Visual Studio does not try to extract these .zip files every time the New Project or Add New Item dialog boxes are shown. Checking a folder that is changed infrequently would incur a performance hit. So to force Visual Studio to install a template for all users, after copying the file into the appropriate folder, you need to run the command *ProgramName/setup* from a command prompt, where *ProgramName* is devenv, vbexpress, csexpress, vwdexpress, or another for the appropriate program that you are setting up the template for.

Wizard Extensions

Although the template wizard does all of the work necessary to process a .vstemplate file, there might be times when you need to customize how a project or project item is generated. For example, suppose you need to display UI to the user to configure how the template is generated, or maybe you need to copy some files into the global assembly cache (GAC)

before the project is created so that the project will run correctly, or maybe you need to modify one of the replacement parameter values to your own specification before the .vstemplate file is processed. With a wizard extension, you can easily add to the template wizard the ability to run your own custom code at opportunistic times when the project, the project item, or the group project is being created.

To create a wizard extension, you will need a class library that implements a specific interface, and information about the wizard extension needs to be added to the .vstemplate file. The definition of this interface is contained within the assembly Microsoft.VisualStudio.TemplateWizard.dll. The methods of this interface, which you need to implement, are as follows:

- ***void RunStarted(object automationObject, System.Collections.Generic.Dictionary<string, string> replacementsDictionary, Microsoft.VisualStudio.TemplateWizard.WizardRun Kind runKind, object[ ] customParams)*** This method is called just after your wizard extension has been loaded and before the TemplateData section of the .vstemplate file starts to be processed. This method provides you with data such as the automation model (an instance of a DTE object) of the application running the wizard passed through the applicationObject parameter. The runKind parameter provides you with an enumerated value that can be AsNewItem if the wizard is being invoked to add a new item to an existing project, AsNewProject if a new project is being created, or AsMultiProject if a group project is being added to a solution. The customParams argument provides a way for the host application to pass context-sensitive information to the wizard extension, but this array usually contains 0 elements. But the replacementsDictionary is the argument that gives you the most power over how the template wizard processes a .vstemplate file. Table 4-2 listed replacement values that the template wizard will search for in files within the project or project item that is being created. The replacementsDictionary contains a list of the tokens to replace and the values that will be used to replace with. This Dictionary object can be read from and written to, meaning that you can modify, add, or remove the values that replacements will be made with. The RunStarted method is also a good place for you to display any UI that might be necessary for configuring your template; you can combine any user input from UI to modify the dictionary and control how the template is rendered.
- ***void BeforeOpeningFile(EnvDTE.ProjectItem projectItem)*** This method is called just before a file is opened. The OpenInEditor="true" attribute on the ProjectItem tag must be specified for the item to be opened.
- ***void ProjectFinishedGenerating(EnvDTE.Project project)*** This method is called when all processing to create a project is complete, the project has been loaded into the solution, and the project is open. You can use the automation model for the project to do any further manipulations that might be necessary. This method is called only when a new project or project group is being created; it will not be called for Add New Item templates.

- ***void ProjectItemFinishedGenerating(EnvDTE.ProjectItem projectItem)*** ProjectItem FinishedGenerating is similar to the ProjectFinishedGenerating method, except this method is called when an Add New Item template is processed.
- ***bool ShouldAddProjectItem(string filePath)*** This method is called when an Add New Item template is processed, and it is supplied the file path of the item that is being generated. If you return true from this method, the item will be added to the project, and if you return false, the item will not be added to the project. This method is useful for wizards that show UI, and, based on the input from the UI, this method selectively adds files to a project. The Visual Web Developer Add New Item templates use this method for the Place Code In Separate File check box in the Add New Item dialog box. The new Web Service template has three project item files listed: WebService.asmx, WebService_cb.asmx, and CodeBehind.cs. If the Place Code In Separate File check box is selected, this method returns false for the WebService.asmx file, while the value false is returned for WebService_cb.asmx and CodeBehind.cs files if the check box is not selected.
- ***void RunFinished()*** After all processing of a .vstemplate file is complete, the wizard will call this method allowing you to perform any cleanup your code needs to do.

After you have created your wizard extension, you must place the assembly implementing the extension in a place where Visual Studio can find and load it. For security reasons, the assembly must be in either the directory containing the executable for that application (such as the folder containing Devenv.exe), a subdirectory containing the executable, or a directory listed in the .config file for the executable (such as Devenv.exe.config) in the probing section of the XML. This security restriction is in place because you do not want templates installed through the Internet with the Content Installer to be able to execute code. Suppose you were to download template content from the Internet and create a project from that template. If the VSTemplate .zip file contained an assembly implementing *IWizard*, and the .vstemplate file referenced that assembly, when the project is created it will cause that code to run. You can see how this is an easy way for a malicious hacker to place code on your computer, and the innocent action of creating a project would run that code. Templates from the Internet can still run code through a wizard extension, but if you need to have elevated permissions (such as Administrator permission) to install an assembly into one of the special directories from which an assembly will be loaded, you minimize the risk of installing bad templates, but you still provide the ability to run wizard extensions. This also means that if you are creating wizard extensions, you need to make sure that your *IWizard* extension cannot be used to cause harm to a user's computer.

Creating a new wizard extension is easy with the WizardExtension starter kit. This starter kit, which is available with the samples for this book, will create a C# class library that implements the *IWizard* interface and create a fragment of XML that you can paste into a .vstemplate file. All you need to do is supply the .vstemplate and copy the .dll file into the correct location so that it can be loaded.

Security Attributes

You can strengthen the security of a wizard extension through the use of attributes. Much as you can place attributes into your *IImportCommunityContent* implementation of a custom installer to restrict which content can be installed, you can place attributes on your class implementing *IWizard* to restrict which templates can call into your assembly. The *TemplateWizardDisallowUserTemplatesSecurityAttribute* attribute takes a Boolean value. If this value is true, then only templates that are installed into the Program Files\Microsoft Visual Studio 8\... location can call into your wizard extension, and templates installed into the My Documents location cannot load and call the extension. If this value is false (the default), any template, regardless of where it is stored, can call into the wizard extension. Another attribute, *TemplateWizardSecurityAttribute*, limits which template can call into your wizard extension. This attribute takes a string that is the file name without full path information, but with the .vstemplate extension of a template that can call the wizard extension. When the template wizard loads the wizard extension, it first checks for the *TemplateWizardSecurityAttribute*. If it is found, it compares the string passed to the attribute with the file name of the template. If the two strings match, the wizard extension will be loaded and run. If it does not match, the wizard will not be loaded. Multiple *TemplateWizardSecurityAttribute* attributes can be placed on the class implementing the wizard extension so that you can call the extension from multiple templates. If you are using the *TemplateWizardSecurityAttribute* attribute, you should also be using the *TemplateWizardDisallowUserTemplatesSecurityAttribute* attribute. If you were to use only the *TemplateWizardSecurityAttribute* attribute and you specified the template name as MyVBTemplate.vstemplate (with the intention to restrict calling the extension from within the Program Files\Microsoft Visual Studio 8\... location), the user could download from the Internet a template that contains the file MyVBTemplate.vstemplate and then your wizard extension would run. By using these two attributes together, you will make sure that the only templates that will run are those the wizard extension was built to aid.

Looking Ahead

Now that we have shown you how to use some of the features of Visual Studio, we can begin exploring how you can customize Visual Studio programmatically with macros.

Chapter 5

Using Visual Studio Macros

In this chapter:

Macros are automation scripts that you can write to take advantage of functionality built into the integrated development environment (IDE). In this chapter, we'll introduce you to a few of the different macro tools in Microsoft® Visual Studio®. The macros engine in Visual Studio lets you record macros to play back later. You can access macros for playback through a window in the IDE called the Macro Explorer. You can edit and debug your macros in a special IDE called the Macros IDE. We'll show you how to record and play macros in Visual Studio and how to write and edit macro projects in the Macros IDE. Finally, we'll also show you how you can share your macros with others.

Macros: The Duct Tape of Visual Studio

The macros facility of Visual Studio uses Microsoft Visual Basic® as its macro language. The Visual Basic language can take full advantage of the Microsoft .NET Framework and its own automation object model, so it offers an extremely powerful and compelling set of features that you can use to automate tasks in the IDE.

Visual Studio macros are saved in files with the .vsmacros extension. These macros are stored in the VSMacros80 folder in your default Visual Studio projects folder. You can specify the Visual Studio projects folder in the Options dialog box, which is on the Projects and Solutions page in the Environment folder. By default, the path to this folder is My Documents\Visual Studio Projects.

Visual Studio macros are usually created in one of two ways. You can record a macro in the IDE (Ctrl+Shift+R); the code generated during the recording session will be stored in the *MyMacros.RecordingModule.TemporaryMacro* method by default. Alternatively, you can open the Macros IDE (Alt+F11) and create a new method by writing it from scratch. One of the best things about macros is that they're designed to automate functionality in the Visual Studio IDE. This means you can often simply record a macro, copy the generated code to a new method, and then use that as the basis for your own automation project.

Visual Studio macros are accessed in the IDE just like any other named command. You can enter the name of the macro in the Command Window (Ctrl+Alt+A), you can add the macro to a toolbar or a menu, you can assign the macro a keystroke shortcut, you can run the macro by double-clicking it in Macro Explorer, and you can run the macro directly from the Macros IDE.

Note When you run a macro by double-clicking it in the Macro Explorer window, the focus returns to the last active window. As a result, you can set the active document, open Macro Explorer, and then double-click the macro to have it affect the last active document.

We consider macros the "duct tape" of Visual Studio—in the best sense of the term. Duct tape is made of an extremely strong material and can help you accomplish tasks quickly and easily. We would describe macros in the same way—they're extremely powerful tools in the IDE that you don't have to spend a ton of time thinking about. You can create your macro to perform your task and then tuck it away. If the macro is sufficiently important and powerful, you can later turn it into a full-blown add-in and then polish that code as much as you want.

Recording Visual Studio Macros

To record a Visual Studio macro, first press the Ctrl+Shift+R keyboard shortcut. This combination brings up the Recorder toolbar and creates a macros module named *RecordingModule* if one doesn't already exist. You can see the Recorder toolbar in Figure 5-1. Notice that you can pause, stop, or even cancel the recording session that you've started.

Figure 5-1 The Recorder toolbar

The easiest way to get going with macros is to record a simple macro that you might want to use repeatedly. For example, let's say you want to find the word *ItemListView* in your code files. You would normally use the Find or Find In Files command for this purpose. But by using one of these commands in the context of a macro, you gain more flexibility and can use the macro in later sessions.

Here are the steps for recording the macro we have in mind:

1. Press Ctrl+Shift+R to start the macro recorder.
2. Press Ctrl+F to open the Find and Replace dialog box.
3. Type ItemListView in the Find What box.
4. Click Find Next.
5. Press Ctrl+Shift+R to stop recording.

We now have a *TemporaryMacro* method saved in the module *RecordingModule*. You can see that macro in Figure 5-2.

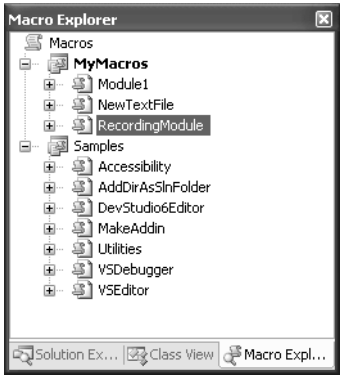


Figure 5-2 The Macro Explorer window

Here's the code that's generated by the preceding procedure. Notice that mouse movements and keystrokes (such as Tab for navigating to the Replace dialog box) aren't recorded. Visual Studio limits macro recording to actual named commands that are called during the recording session.

```
Option Strict Off
Option Explicit Off
Imports EnvDTE
Imports EnvDTE80
Imports System.Diagnostics
Public Module RecordingModule
    Sub TemporaryMacro()
        DTE.ExecuteCommand("Edit.Find")
        DTE.Windows.Item("ScreenSaverForm.vb").Activate()
        DTE.Find.FindWhat = "ItemListView"
        DTE.Find.Target = _
vsFindTarget.vsFindTargetCurrentDocument
        DTE.Find.MatchCase = False
        DTE.Find.MatchWholeWord = False
        DTE.Find.Backwards = False
        DTE.Find.MatchInHiddenText = True
        DTE.Find.PatternSyntax = _
vsFindPatternSyntax.vsFindPatternSyntaxLiteral
        DTE.Find.Action = vsFindAction.vsFindActionFind
        If (DTE.Find.Execute() = _
vsFindResult.vsFindResultNotFound) Then
            Throw New System.Exception("vsFindResultNotFound")
        End If
    End Sub
End Module
```

To play back this macro, press Ctrl+Shift+P, which is simply a shortcut to the Macros.Macros.RecordingModule.TemporaryMacro command. You should see the Find dialog box

open with the first instance of the word you're searching for selected. In our case, this is the first instance of *ItemListView* in a file named Connect.cpp.

Take a look at the line *DTE.Windows.Item(˝ScreenSaverForm.vb˝).Activate()*. If ScreenSaverForm.vb isn't already open, this line will bring it into focus in the IDE, so this macro won't be very useful if you want to save it for use with a number of different files or projects. Commenting out or removing this line from the code will cause the macro to work with the currently active document.

To save the recorded macro, you can either rename *TemporaryMacro* to something else in Macro Explorer or you can copy and paste the recorded code into another macro module or method.

Macro Commands

Macro Explorer lets you manage your macros from inside the Visual Studio IDE. You can access the commands related to macros in the IDE from the Macros submenu of the Tools menu or through the shortcut menus within Macro Explorer.

Macros are divided into projects containing modules, which in turn contain methods. Projects are represented hierarchically in Macro Explorer below the Macro icon. Right-clicking the Macro icon brings up the shortcut menu containing commands for creating and loading macro projects. You can access the same functionality via named commands in the Command Window. Table 5-1 lists the macro commands related to macro projects.

Table 5-1 Macro Project Commands

Command	Description
Tools.LoadMacroProject	Brings up the Add Macro Project dialog box, in which you can select a macro project file.
Tools.NewMacroProject	Brings up the New Macro Project dialog box, in which you can save your macros into specific projects.
Tools.MacrosIDE	Brings up the Macros IDE. This command is mapped to Alt+F11.

You can open Macro Explorer by pressing Alt+F8. Most commands available from the shortcut menus in Macro Explorer are also available from the Command Window (because the items in Macro Explorer lose focus when you change to the Command Window). You can rename a macro project by right-clicking on the project in Macro Explorer and then clicking Rename. Doing so will allow you to edit the name of the macro project in place. You can delete a macro project by choosing Delete from the shortcut menu. The same basic shortcut menu items are available for renaming and deleting modules and methods from within Macro Explorer.

By right-clicking on a macro in Macro Explorer, you can bring up a shortcut menu that lets you work with the macro directly. The Run command executes the Tools.Run command on the currently selected macro. The Rename command allows you to edit the name of the macro in place. The change you make to the name is reflected in the method name in the Macros IDE. The Delete command deletes the currently selected macro. And finally, the Edit command opens the current macro in the Macros IDE.

For organizing the macros you've created, Macro Explorer is a powerful tool. You'll find that you can do quite a bit in Macro Explorer. For example, you can record a macro, rename that macro to save it, and even add that same macro to a toolbar or a menu in the IDE, all without having to go to the Macros IDE. That said, to really get the most out of Visual Studio macros, you'll want to be able to create and edit them from within the Macros IDE.

Editing Macros in the Macros IDE

Working with the Macros IDE is similar to working in Visual Studio. Many of the same shortcuts work in the Macros IDE. The Macros IDE editor features IntelliSense®, and the Help system for macros is integrated into the IDE.

One difference you'll notice right away is that all your loaded macro projects show up in the Project Explorer window. Visual Studio ships with an extremely useful set of macros out of the box. You can see these macros if you expand the Samples project in Project Explorer in the Macros IDE (as shown in Figure 5-3).

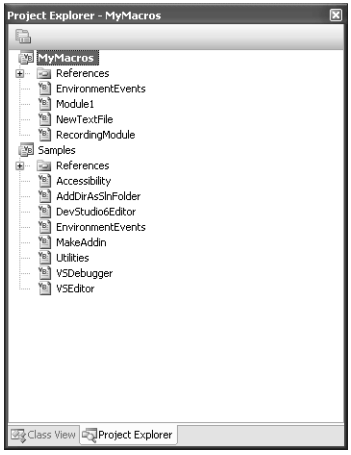


Figure 5-3 The Samples project in the Macros IDE

The memory space for macro projects is separated, so if you want to utilize functionality between different macros or if you want to take advantage of a common set of environmental events, you must keep the macros that you write inside the same project. If you want to access functionality from another macro project, you can simply copy the macros you want

to access into the project you're working on. For example, you can copy modules from the Samples project into your own project to take advantage of the functionality exposed by those macros.

To create a new macro project, you can right-click in the Macro Explorer window, and then click New Macro Project from the shortcut menu to open the New Macro Project dialog box (shown in Figure 5-4). Enter a name and location for your project, and then click OK. Pressing Alt+F11 will toggle you back to the Macros IDE, where you can work on the code in the new project.

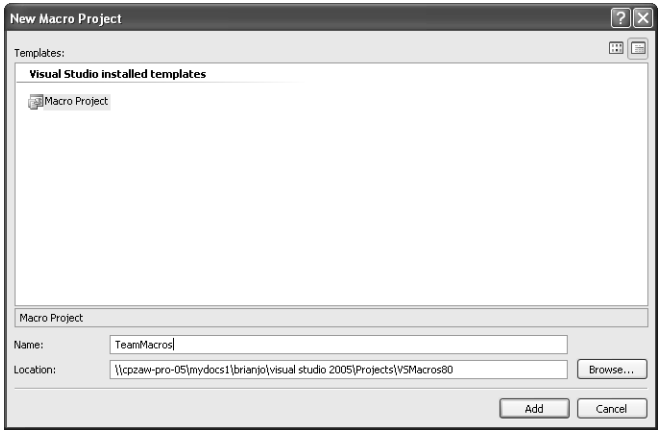


Figure 5-4 The New Macro Project dialog box

If you take a look at the new macro project created in Project Explorer, you'll notice that a number of features are added to your project by default. The References folder works similarly to the References folder in the Visual Studio IDE. Two new modules are added to get your macros up and running: the *EnvironmentEvents* module contains generated code that gives you access to the events in the IDE, and the *Module1* module provides a place where you can start writing code.

Adding a reference to a macro project is slightly different from adding one to a standard Visual Basic project. If you look at the Add Reference dialog box that's used in the Macros IDE Project Explorer (shown in Figure 5-5), you'll notice that it doesn't offer a way to add custom assemblies.

To add references to your own assemblies, you must copy them to the C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\PublicAssemblies folder. You can then add your own reference to the assembly from the Add Reference dialog box. Using assemblies, you can write your macro functionality in any language you want and then access that functionality from a fairly simple macro. You can also write assemblies that call to unmanaged code and assemblies that act as COM wrappers to access COM functionality from within your macros.

Let's go over a few examples built from a new project.

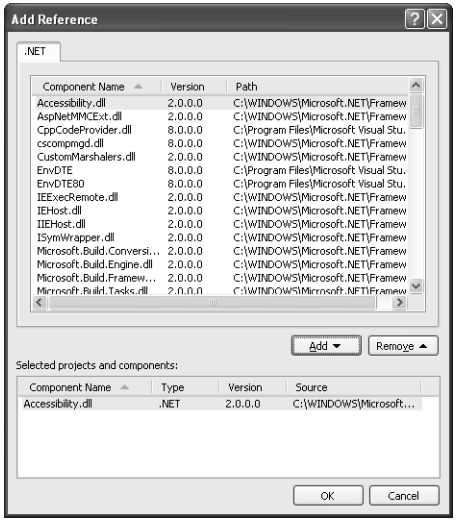


Figure 5-5 Add Reference dialog box

A Simple Macro

The File.NewFile command in Visual Studio opens a dialog box that allows you to choose the type of file that you want to create. Some programmers would rather have a command like this generate a new file than present a dialog box. The solution is simple: you just create a macro that does exactly what you want and then assign that macro an alias in the Command Window. The following code is all you really need to create a new text file in the IDE:

```
Imports EnvDTE
Imports System.Diagnostics

Public Module NewFile

    Sub NewTextFile()
        DTE.ItemOperations.NewFile("General\Text File")
    End Sub

End Module
```

As you can see, this macro has been created in a module named *NewFile*. It consists of a single method, *NewTextFile*. The single line of code in this macro simply creates a new file of the type Text File in the General folder of the New File dialog box. We'll talk about the *NewFile* method that creates the new text file shortly. What's important right now is that we have a macro that will add just the functionality we want to the IDE. To make this macro a tool we're willing to spend some time with, we'll want to make the macro as easy to access as possible.

To access a macro you want to execute, you have a few choices. One approach is to run the macro from Macro Explorer in the Visual Studio IDE. This works fine, but it's probably not

the optimal solution for a macro that you're planning to use often. The second choice is to create an alias for the macro in the Command Window. This is probably the best choice for a command that you want to use while you're typing. To alias this command, you type **alias** followed by the command name, followed by the name of the macro. IntelliSense will kick in when you start to type a macro, so the whole alias line might look something like this:

```
>alias nf Macros.MyMacros.NewTextFile.NewTextFile
```

Now you have a new command, nf, that you can use from the Command Window. To create a new text file, you can simply press Ctrl+Alt+A and then type nf to get your new file. Of course, if you want to take it a step further, you can assign the macro a keystroke shortcut from the Options dialog box. In keeping with the Ctrl+0 initial sequence introduced in Chapter 3, Ctrl+0, Ctrl+N might make a good shortcut. Finally, you can add a button to the toolbar that initiates the macro (as described in Chapter 3).

The *Imports* statement in this sample is important. The API associated with the Visual Studio automation object model is contained in the *EnvDTE* namespace. The automation object model is discussed in depth throughout the rest of the book. Here we simply want to familiarize you with this object model and get you up and running with some of the more common functionality that you'll use in your macro projects. Most of the subjects covered in the remainder of the book apply to both macros and add-ins. In fact, you can use macros to test code that you want to write into your add-ins. You'll save time because you normally test an add-in by compiling the add-in and loading a second instance of the IDE. Using a macro, you can get to the automation object model, write and test your routines, and then add them to your add-in projects.

Working with Macros

The macros you build will use the automation object model to access and automate the different parts of the IDE. In this section, we'll demonstrate how you can use macros to automate some simple tasks, and we'll talk a bit about the automation object model as it applies to documents and windows in the IDE. We'll also discuss events and provide some simple examples to help you get going right away.

Manipulating Documents and Text

Some of the most useful tasks you can perform with macros involve working with text in documents. You might want to search for text, change a selection in some way, or just insert text into a document. The *Document* object in the *DTE* provides a good deal of functionality that makes it easy to manipulate text in code documents.

Macros are often run on the document with the current focus. To get the currently active document in the IDE, use the *DTE.ActiveDocument* property, which returns a *Document* object. (Recall that a Visual Studio document is an editor or a designer window that opens to the center of the IDE.) If the document is an editor, it has an associated *TextDocument* object.

The *TextDocument* object has three properties of interest for programmers who want to manipulate text inside the object. The *StartPoint* property returns a *TextPoint* object that points to the beginning of the document. The *EndPoint* property returns a *TextPoint* object that points to the end of the document. And finally, the *Selection* property returns a *TextSelection* object that offers a number of properties and methods you can use on selected text.

The *TextPoint* object provides location information for the editing functionality inside a document. You create a *TextPoint* in a document whenever you want to insert or manipulate text in the document or when you want to get some information about a particular document. *TextPoint* objects aren't dependent on text selection, and you can use multiple *TextPoint* objects in a single document.

Let's look at a couple of examples that use the objects we've mentioned. You should become familiar with this code because much of the macro automation code you'll write will depend on it.

First, let's get the *ActiveDocument*, create a couple of *EditPoint* objects, and then add some text to the *ActiveDocument* by using that information:

```
Sub CommentWholeDoc()
    Dim td As TextDocument = ActiveDocument.Object
    Dim sp As TextPoint
    Dim ep As TextPoint
    sp = td.StartPoint.CreateEditPoint()
    ep = td.EndPoint.CreateEditPoint()

    sp.Insert("/* ")
    ep.Insert(" */")

End Sub
```

Running this sample on a Microsoft Visual C#® or a Microsoft Visual C++® code document will comment out the entire document. The macro isn't very practical, but it does show you how to put those parts together. You can use IntelliSense to make your way through the objects created so that you can experiment with some of the other functionality.

Let's take a look at a second, more useful, example that inserts text into a document based on a selection. The following example wraps selected text with whatever text we want in a document. Here we'll declare *ts* as a *TextSelection* object and assign it the current selection using *DTE.ActiveDocument.Selection*:

```
Sub HTMLComment()
    Dim ts As TextSelection = DTE.ActiveDocument.Selection
    Dim ep As TextPoint
    ep = ts.BottomPoint.CreateEditPoint()
    ts.Insert("<!-- ", vsInsertFlags.vsInsertFlagsInsertAtStart)
    ep.Insert(" -->")
    ts.Collapse()
End Sub
```

This macro uses the *TextSelection Insert* method to insert text at the beginning of the *Selection* object. The *Insert* method takes two arguments. The first argument is the string that you want to insert into the selection. The second argument is a *vsInsertFlags* constant that defines where the insertion is to take place. The *Insert* call in the example uses *vsInsertFlagsAtStart*. To close the tag, we use an *Insert* with a *TextPoint* because the insertion point changes after the first insert. If we just wanted to add text to the end of a selection, we could just use *vsInsertFlagsInsertAtEnd*. Table 5-2 lists these constants.

Table 5-2 *vsInsertFlags* Constants

Constant	Description
vsInsertFlagsCollapseToStart	Collapses the insertion point from the end of the selection to the current TextPoint
vsInsertFlagsCollapseToEnd	Collapses the insertion point from the beginning of the selection to the current TextPoint
vsInsertFlagsContainNewText	Replaces the current selection
vsInsertFlagsInsertAtStart	Inserts the text before the start point of the selection
vsInsertFlagsInsertAtEnd	Inserts text just after the end point of the selection

With a *Selection*, a *TextPoint*, and the methods available through the *DTE*, you should have a good basis for the types of operations you can perform on source code by using macros.

Moving Windows

Windows in Visual Studio are controlled through the *Window* object, which is part of the *DTE.Windows* collection. The *Window* object provides functionality based on the window type. Specifically, the *CommandWindow*, *OutputWindow*, *TaskList*, *TextWindow*, and *ToolBox* derive from the *Window* object.

Of the window objects, *OutputWindow* is among the most practical for macro writing. You can use it to display and hold messages in much the same way you would use *printf* or *Console.Write* in a console application or in the same way that you use *MsgBox* or *MessageBox.Show* in a Microsoft Windows®–based application.

To use the *OutputWindow* object to display messages, you must create a new method that takes a string argument. You can then call the method with the argument in the same way you use the *MsgBox* method to display a message. The following example is a method named *MsgWin*. It takes only a string as an argument. You can use this method in place of *MsgBox* when you want to see a bit of text information quickly.

```
Sub MsgWin(ByVal msg As String)
    Dim win As Window = DTE.Windows.Item(Constants.vsWindowKindOutput)
    Dim cwin As Window =
        DTE.Windows.Item(Constants.vsWindowKindCommandWindow)
    Dim ow As OutputWindow = win.Object
    Dim owp As OutputWindowPane
    Dim cwp As CommandWindow = cwin.Object
    Dim i As Integer
    Dim exists As Boolean = False
    ' Check to see if we're running in the Command Window. If so,
    ' we'll send our output there. If not, we'll send it to a Command
    ' window.
    If (DTE.ActiveWindow Is cwin) Then
        cwp.OutputString(msg + vbCrLf)
    Else
        ' Determine if the output pane name exists. If it does, we need
        ' to send our message there, or we end up with multiple windows of
        ' the same name.
        For i = 1 To ow.OutputWindowPanes.Count
            If ow.OutputWindowPanes().Item(i).Name() = "MsgWin Output" Then
                exists = True
                Exit For
            End If
        Next
        ' If our output pane exists, we'll use that to output the string,
        ' otherwise, we'll add it to the list.
        If exists Then
            owp = ow.OutputWindowPanes().Item(i)
        Else
            owp = ow.OutputWindowPanes.Add("MsgWin Output")
        End If
        ' Here we set the Output window to visible, activate the pane,
        ' and send the string to the pane.
        win.Visible = True
        owp.Activate()
        owp.OutputString(msg + vbCrLf)
    End If
End Sub
```

To use the *MsgWin* macro, you must call it from another method. For this example, we've created a method that lists all the currently open windows in the IDE:

```
Sub MsgWinTest()
    Dim wins As Windows = DTE.Windows()
    Dim i As Integer

    For i = 1 To wins.Count
        MsgWin(wins.Item(i).Caption.ToString())
    Next
End Sub
```

Figure 5-6 shows what the Visual Studio IDE looks like after it has been invoked from the *MsgWinText* macro in the IDE.

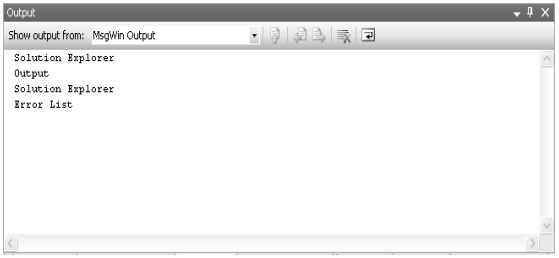


Figure 5-6 The MsgBox Output window in the IDE

You can do a lot of things with this basic *MsgWin* macro to improve it. It would be pretty trivial to overload the *MsgWin* method to allow for such actions as clearing the output pane or adding a heading to the list. For example, to create an overload for the *MsgWin* function that clears the output pane, you can make the method look something like this:

```
Sub MsgWin(ByVal msg As String, ByVal clr As Boolean)
        §
        ' If clr is True then we'll clear the output pane.
        If clr = True Then
            owp.Clear()
        End If
        ' Here we set the Output window to visible, activate the pane,
        ' and then send the string to the pane.
        win.Visible = True
        owp.Activate()
        owp.OutputString(msg + vbCrLf)
    End If
End Sub
```

Macro Events

One of the most powerful features of macros in the IDE is an event model that lets you fire macros based on events that take place in the IDE. You can use events to fire macros that create logs, reset tests, or manipulate different parts of the IDE in the ways we've already talked about in this chapter. In this short section, we'll show you how to create event handlers for different events in the IDE. Using this information and the detailed information about the different parts of the automation API discussed throughout the rest of the book, you should have a good idea of how to take advantage of events in your own projects.

The easiest way to get to the event handlers for a macros project is through the Project Explorer window in the Macros IDE. Expand a project, and you'll see an *EnvironmentEvents* module listed. Open that file, and you'll see a block of code that's been generated

automatically by the IDE. Here's the important part of the block (the attributes have been removed to make this fit the page):

```
Public WithEvents DTEEvents As EnvDTE.DTEEvents
Public WithEvents DocumentEvents As EnvDTE.DocumentEvents
Public WithEvents WindowEvents As EnvDTE.WindowEvents
Public WithEvents TaskListEvents As EnvDTE.TaskListEvents
Public WithEvents FindEvents As EnvDTE.FindEvents
Public WithEvents OutputWindowEvents As EnvDTE.OutputWindowEvents
Public WithEvents SelectionEvents As EnvDTE.SelectionEvents
Public WithEvents BuildEvents As EnvDTE.BuildEvents
Public WithEvents SolutionEvents As EnvDTE.SolutionEvents
Public WithEvents SolutionItemsEvents As EnvDTE.ProjectItemsEvents
Public WithEvents MiscFilesEvents As EnvDTE.ProjectItemsEvents
Public WithEvents DebuggerEvents As EnvDTE.DebuggerEvents
```

As you can see from this code, there are a lot of event types you can take advantage of in the IDE. In fact, you can use all the DTE events, even though they're not included by default. You can add these other events to this list to get to the events that you're interested in. To create a new event handler, you select the event type you want to handle from the class name list at the top of the code window. You can see how this looks in Figure 5-7.

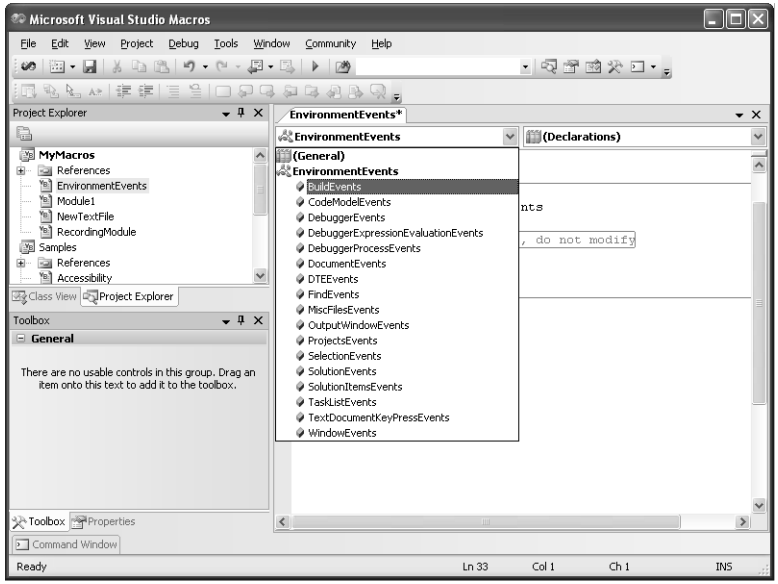


Figure 5-7 Selecting the event type you want to handle from the class name list

After you select an event type, the method name list in the upper-right portion of the code pane will list the events you can handle, as shown in Figure 5-8.

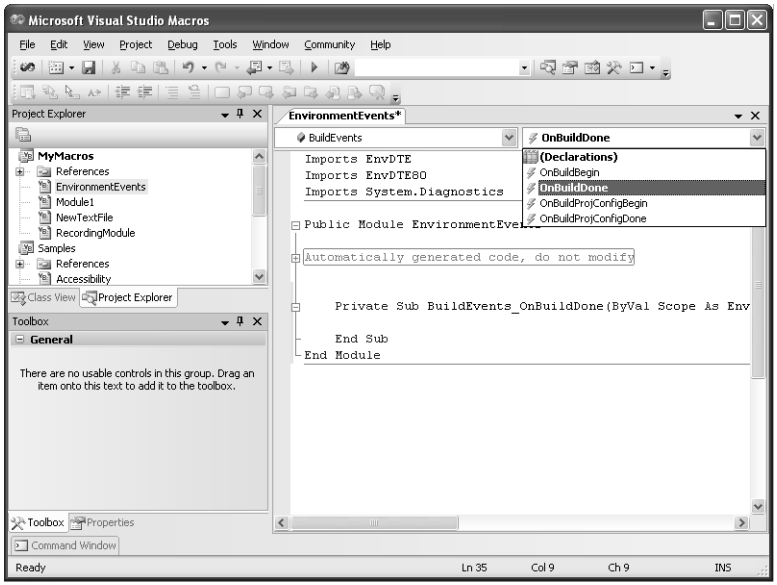


Figure 5-8 Selecting the event you want to handle from the method name list

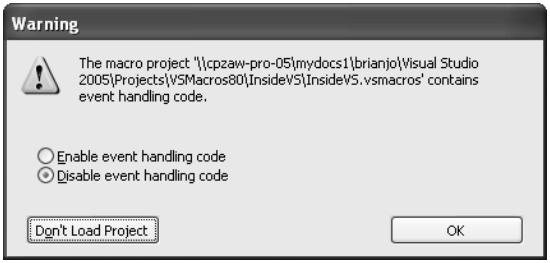
Select the event you want from the list, and your event handler will be generated automatically. From this generated event handler, you can call a method that you've created in the project, or you can add your event-handling functionality directly to the event-handler code. In this example, we'll call the *MsgWin* function that we worked through earlier to display a message that indicates that the build has completed.

```
Private _
Sub BuildEvents_OnBuildDone(ByVal Scope _
   As EnvDTE.vsBuildScope, _
   ByVal Action As EnvDTE.vsBuildAction) _
   Handles BuildEvents.OnBuildDone
        MsgWin("Build is done!")
End Sub
```

As you can imagine, these events open up all sorts of possibilities for automation and customization in the IDE. One thing you should keep in mind when working with events is that all the code in a single macro project shares the same event module. This means that if you want to create different event handlers for the same event, you'll need to create the other event handlers in other projects.

Event Security

As you can imagine, executing event code in a powerful macros facility such as the one in Visual Studio has some potential security implications. The first time you load a macro project that contains event-handling code, you see a dialog box that looks like this:



You should be sure you know where your macros come from when you load macro projects. If you're not sure of the event-handling code in the project, click Disable Event Handling Code in the Warning dialog box and review the code in the module before you use it.

Sharing Macros with Others

If you want to share the macros that you've created, you have a number of choices to make. Do you want to share the source? Do you want to share the whole project or just part of it? The answers to these questions will determine how to best share your work. Let's take a look at the different ways that you can share your macro functionality with others.

Exporting Modules and Projects

The easiest way to share your macros with other developers is to simply cut and paste your source code into e-mail messages and Usenet postings. This approach works well if the methods you're sharing are fairly short and if they don't span multiple modules. If they do span multiple modules, you'll probably want to export the modules you want to share or simply pass on the whole project.

To export a macro module in Visual Studio, you must open the Macros IDE and select the module you want to export from the Project Explorer window. Pressing Ctrl+E will invoke the File.SaveSelectedItemsAs command, which brings up the Export File dialog box. This command is listed on the File menu as the Export *(module name)* command.

The Export File dialog box lets you save the module as a .vb file that you can easily import into another project by using the File.AddExistingItem command (Shift+Alt+A). Don't forget to include the code from the *EnvironmentEvents* module if your macros rely on some sort of event functionality.

If your macros are very complicated, you might want to share an entire macro project. You can do this in a couple of ways. You can copy the .vsmacros file for the project and pass it along, or you can save your macro project as a text-based project and share those files.

To make a macros project text-based, change the *StorageFormat* property in the Visual Studio IDE for the project that you want to change. Select the project in Macro Explorer and then change the *StorageFormat* property in the Properties window from Binary (.vsmacros) to Text (Unicode). This change will create a number of files in the macro project's folder, which looks much like a regular Visual Studio project folder. In Figure 5-9, you can see the folder for the Samples project after it has been converted to Text format.

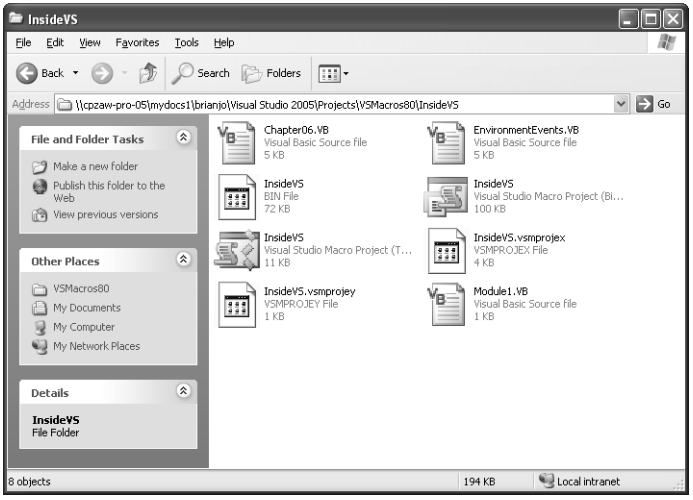


Figure 5-9 A macro project that has been stored in Text format

The advantage of passing along a text-based project is that it allows other programmers to look at the source files in your project before loading them into their IDEs.

There's always a security risk in opening unknown macro projects in any application. Be sure you know where any binaries you open come from. At the very least, check the EnvironmentEvents.vb module to make sure it doesn't include any unexpected code.

Looking Ahead

This chapter gave you some information about using the Visual Studio macro facility to perform some simple automation tasks. The remainder of the book should provide you with enough information to fully utilize automation in Visual Studio from macros or add-ins.

Chapter 6

Extending the IDE with Add-Ins

In this chapter:

You learned in Chapter 5 that macros provide a convenient way to automate tasks within Microsoft® Visual Studio®, and we encourage you to write macros first when customizing the integrated development environment (IDE). But for some purposes, such as writing commercial software, you might find that macros are a poor choice in terms of performance and protection of intellectual property. In such cases, the appropriate vehicle is an *add-in*, which is a compiled dynamic-link library (DLL) (providing increased protection) that runs within the IDE (providing increased performance).

The quickest way to get started as an add-in programmer is through the Add-in Wizard, which makes creating add-ins easy—just choose a few options, and the wizard generates an add-in that is ready for the IDE. Of course, you won't accomplish much if you don't know how to control and customize the add-in, so in the first half of this chapter, we'll show you how the different parts of an add-in act together to make it work.

The second half of this chapter covers the details of add-in architecture. In this part of the chapter, we'll hold to the ideal that wizards are tools, not crutches, and that you should use them as a convenience only after you're capable of writing the equivalent code. Of course, we don't expect you to reach that goal without a little help, so we'll teach you everything you need to know to write the equivalent of a wizard add-in.

Running the Add-In Wizard

When you choose File | New | Project, Visual Studio offers its selection of project types in the New Project dialog box. By expanding the Other Projects node and selecting Extensibility, you'll find the Visual Studio Add-in template shown in Figure 6-1; double-click its icon to launch the Add-in Wizard.

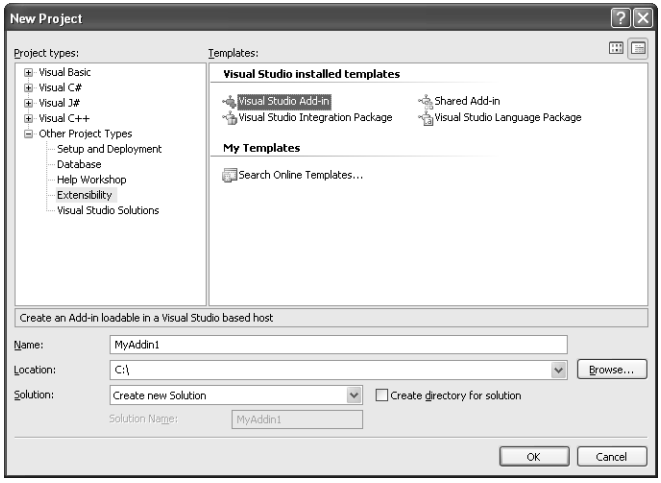


Figure 6-1 The Visual Studio Add-in template

The six pages of the Add-in Wizard collect your choices about the final form of your add-in. The wizard gives you control over the following areas:

- **Programming language** The Add-in Wizard generates the add-in source code in one of four programming languages—Microsoft Visual C#®, Microsoft Visual Basic® .NET, Microsoft Visual J#®, or Microsoft Visual C++® (by using either Managed C++ or the Active Template Library [ATL]). Of course, you're not restricted to these languages when you write add-ins by hand.
- **Application host** Add-ins can run in the Visual Studio IDE, the Macros IDE, or both. With few exceptions, the rules that apply to an add-in running in the Visual Studio IDE also apply to an add-in running in the Macros IDE. (We'll point out differences between the two hosts when appropriate.)
- **Name and description** These settings let you associate a meaningful name and description with your add-in.
- **Menu command** The Add-in Wizard can generate code that creates a new menu item for your add-in, giving users a convenient way to load your add-in and execute a command.
- **Command-line build support** You can mark your add-in as being safe for use with unattended builds. Such an add-in promises that it won't display user interface elements that require user intervention (such as modal dialog boxes).
- **Load at startup** Add-ins can request that they be loaded automatically when Visual Studio starts up.
- **About box information** You can provide support information for your add-in that Visual Studio will display in its About dialog box.

When the Add-in Wizard finishes, it generates an add-in project that builds the add-in DLL.

The Add-In Project

Add-ins are DLLs, so the Add-in Wizard creates a Class Library project for your add-in. This project contains a source file named Connect, which defines the add-in class, also named *Connect*. The *Connect* class implements the *IDTExtensibility2* interface, which serves as the main conduit for add-in/IDE communication. (*Connect* also implements *IDTCommandTarget* if you select the user interface option in the Add-in Wizard.) Table 6-1 lists the five methods of the *IDTExtensibility2* interface.

Table 6-1 ***IDTExtensibility2*** **Interface Methods**

Method	Description
OnConnection	Called when the add-in is loaded
OnStartupComplete	Called when Visual Studio finishes loading
OnAddInsUpdate	Called whenever an add-in is loaded or unloaded from Visual Studio
OnBeginShutdown	Called when Visual Studio is closed
OnDisconnection	Called when the add-in is unloaded

The Connect.cs file in Listing 6-1 shows the code (minus some comments) that the Add-in Wizard generates for a typical C# add-in with a menu command. We'll walk through the source code, pointing out any interesting features along the way.

Listing 6-1 Connect.cs, the add-in source code generated by the Add-in Wizard

```
using System;
using Extensibility;
using EnvDTE;
using EnvDTE80;
using Microsoft.VisualStudio.CommandBars;
using System.Resources;
using System.Reflection;
using System.Globalization;

public class Connect : Object, IDTExtensibility2, IDTCommandTarget
{
    public Connect()
    {
    }

    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)
        {
            // Generate the add-in's menu item...
        }
```

```
    }

    public void OnDisconnection(ext_DisconnectMode disconnectMode,
        ref Array custom)
    {
    }

    public void OnAddInsUpdate(ref Array custom)
    {
    }

    public void OnStartupComplete(ref Array custom)
    {
    }

    public void OnBeginShutdown(ref Array custom)
    {
    }

    public void QueryStatus(string commandName,
        vsCommandStatusTextWanted neededText, ref vsCommandStatus status,
        ref object commandText)
    {
        if(neededText ==
            vsCommandStatusTextWanted.vsCommandStatusTextWantedNone)
        {
            if(commandName == "MyAddin1.Connect.MyAddin1")
            {
                status = (vsCommandStatus)
                    vsCommandStatus.vsCommandStatusSupported |
                    vsCommandStatus.vsCommandStatusEnabled;
            }
        }
    }

    public void Exec(string commandName, vsCommandExecOption executeOption,
        ref object varIn, ref object varOut, ref bool handled)
    {
        handled = false;
        if(executeOption ==
            vsCommandExecOption.vsCommandExecOptionDoDefault)
        {
            if(commandName == "MyAddin1.Connect.MyAddin1")
            {
                handled = true;
                return;
            }
        }
    }
    private DTE2 _applicationObject;
    private AddIn _addInInstance;
}
```

At the top of the listing, you'll see that the Add-in Wizard generates a set of *using* statements for the programmer's convenience. The three most important namespaces in the *using*

statements are *Extensibility*, *EnvDTE*, and *EnvDTE80*; the first namespace defines the types used by *IDTExtensibility2*, and the latter two define the types in the automation object model.

The first method in the listing, *OnConnection*, wins the prize for "most important add-in method." Visual Studio calls this method when it loads the add-in, and it passes the add-in a reference to the root object of the automation object model through the *application* parameter. The code generated by the Add-in Wizard casts the *application* parameter to the *EnvDTE80.DTE2* type and stores the result in a private variable named *applicationObject*. All further interaction between the add-in and the automation object model takes place through the *applicationObject* variable.

Visual Studio also passes the add-in a reference to its corresponding *AddIn* object through the *addInInst* parameter; the add-in stores this reference in a private variable named *addInInstance*.

The rest of the code in *OnConnection* creates an add-in menu command on the Tools menu. (This code is absent if you forgo the user interface option in the Add-in Wizard.) The menu-creation code executes conditionally, depending on the following *if* statement:

```
if (connectMode == ext_ConnectMode.ext_cm_UISetup)
```

The *connectMode* parameter holds a value that describes how the add-in was loaded. For add-ins that create a menu command, Visual Studio passes in the *Extensibility.ext_ConnectMode. ext_cm_UISetup* value the first time the add-in loads after being installed, which signals to the add-in that this is a good time to add its commands to the IDE.

The Add-in Wizard doesn't generate any code in the bodies of the other four *IDTExtensibility2* methods: *OnStartupComplete*, *OnAddinsUpdate*, *OnBeginShutdown*, and *OnDisconnection*. The two *IDTCommandTarget* methods, *QueryStatus* and *Exec*, have some boilerplate code that helps manage the add-in's menu command and menu command clicks, respectively. To handle menu command clicks, you add code to the *Exec* method in the second *if* statement, which begins with

```
if (commandName == "MyAddin1.Connect.MyAddin1")
```

There isn't much code in Connect.cs, even if you've selected every option in the Add-in Wizard, but the code that's there creates a fully functional add-in that you can build on.

Loading the Add-In

After you build the add-in, you need to load it into Visual Studio. The method you use to load the add-in can vary, depending in part on the options you selected in the Add-in Wizard. If you chose to have your add-in load on startup, Visual Studio will load the add-in automatically each time it runs. If you chose to have a user interface item for your add-in,

the next time Visual Studio runs, you'll be able to load the add-in by choosing its command from the Tools menu, as shown in Figure 6-2.

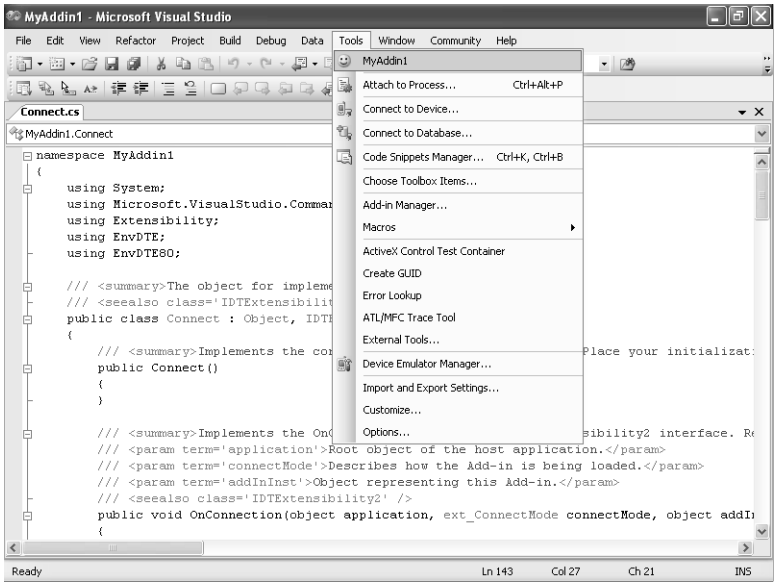


Figure 6-2 A default add-in menu command

If you didn't choose either of these options, you can load the add-in by choosing Tools | Add-in Manager, which launches the Add-in Manager (shown in Figure 6-3). The Add-in Manager gives you control over all the registered add-ins, allowing you to load them, unload them, and mark them to load on startup and during command-line builds.

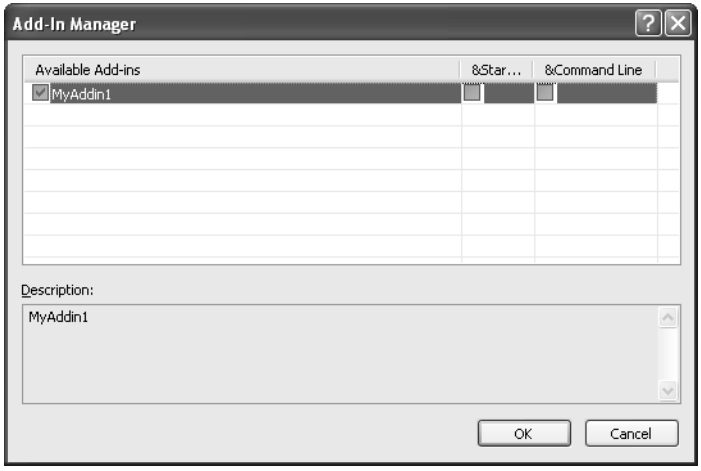


Figure 6-3 The Add-in Manager dialog box

Debugging the Add-In

An add-in is just a DLL, so debugging an add-in project is no different from debugging any other Class Library project. Because a DLL can't run on its own, it needs a host application. For an add-in, that host is Visual Studio (devenv.exe) or the Macros IDE (vsaenv.exe). The Add-in Wizard sets the debugging properties of the add-in project so that Visual Studio is the host. You can examine and modify the project's debugging properties by right-clicking the add-in project in Solution Explorer, choosing Properties from the shortcut menu, and selecting Debug in the Project Designer window shown in Figure 6-4. For most purposes, however, the default settings work just fine.

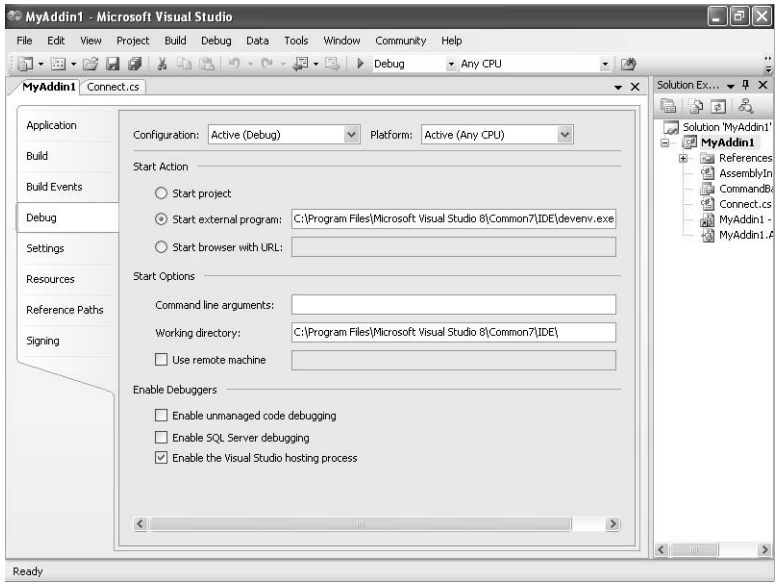


Figure 6-4 The add-in project's debugging properties

In a typical debugging session, you open the add-in project in Visual Studio, set breakpoints in the add-in source code, and then start the debugger by choosing Start from the Debug menu (or pressing F5). The debugger, in turn, launches a second instance of Visual Studio and attaches itself to this new process. You load the add-in to be debugged in the second instance of Visual Studio, and when the add-in code hits a breakpoint, execution passes to the debugger running in the first instance of Visual Studio. From there you can step through the code, examine the contents of variables and registers, and perform other sundry debugging tasks.

Debugging add-ins in the Macros IDE is almost as easy as debugging add-ins in Visual Studio. The one difference is that you can't simply open the Macros IDE from an instance of Visual Studio and then attach the debugger from that instance to the Macros IDE. Why not? Because the two processes will deadlock if Visual Studio fires a macro event while execution

is stopped in the debugger. Instead, the recommended way to debug add-ins in the Macros IDE is similar to the way you debug add-ins in Visual Studio:

1. Open the add-in project in Visual Studio.
2. Start a second instance of Visual Studio.
3. Open the Macros IDE from the second instance of Visual Studio.
4. Attach the debugger from the first instance of Visual Studio to the Macros IDE process. To do so, choose Debug | Attach To Process, select vsaenv.exe in the Attach To Process dialog box, and then click Attach.
5. Load the add-in in the Macros IDE, and then debug as normal.

Finally, you shouldn't feel obligated to run the Visual Studio debugger if you have a favorite debugger you'd rather use. Any debugger that can handle delay-load DLLs will do.

Add-In Architecture

You've learned that the easiest way to create an add-in is by running the Add-in Wizard included with Microsoft Visual Studio. The easiest way isn't always the best way, however, especially when you're trying to learn an unfamiliar technology. It's time to close your IDE and open up a Command Window. In the next section, you'll learn the fundamentals of add-in construction by writing add-ins the old-fashioned way—by hand, from scratch.

Writing an Add-In from Scratch

Listing 6-2 shows the source code for our handwritten add-in named Basic. You can think of Basic as the smallest possible add-in that still does something useful. And as you can see from the listing, the smallest possible add-in is small indeed. That's because add-ins have one requirement only: a public class that implements the *Extensibility.IDTExtensibility2* interface. Basic.cs satisfies this requirement by defining a single public class named Basic that implements the *IDTExtensibility2* interface's five methods—*OnConnection*, *OnStartupComplete*, *OnAddInsUpdate*, *OnBeginShutdown*, and *OnDisconnection*. There's no *Main* method because Basic, as with all add-ins, is destined to become a DLL. Instead, the *OnConnection* method serves as the add-in's entry point, and the Basic add-in implements that method by displaying its own name in a message box.

Listing 6-2 Basic.cs, the Basic add-in source code

```
using System;
using System.Windows.Forms;
using Extensibility;

public class Basic : IDTExtensibility2
{
    public void OnConnection(object application,
        ext_ConnectMode connectMode,
```

```
        object addInInst,
        ref Array custom)
    {
        MessageBox.Show("Basic Add-in");
    }

    public void OnStartupComplete(ref Array custom)
    {
    }

    public void OnAddInsUpdate(ref Array custom)
    {
    }

    public void OnBeginShutdown(ref Array custom)
    {
    }

    public void OnDisconnection(ext_DisconnectMode removeMode,
        ref Array custom)
    {
    }
}
```

Compiling the Basic Add-In

If you add the source code in Listing 6-2 to a text file named Basic.cs, you can compile the Basic add-in from the command line by using the following command:

```
csc /t:library /r:"c:\program files\common files\microsoft
shared\msenv\publicassemblies\extensibility.dll" basic.cs
```

The */t:library* flag directs the C# compiler to create a DLL (Basic.dll) from the source file, and the */r:"c:\program files\common files\microsoft shared\msenv\publicassemblies\extensibility.dll"* flag points the compiler to the assembly that contains the *Extensibility* namespace (Extensibility.dll). The *Extensibility* namespace defines three types, which all add-ins use: the *IDTExtensibility2* interface and the *ext_ConnectMode* and *ext_DisconnectMode* enumerations, which define values passed to the *OnConnection* and *OnDisconnection* methods, respectively.

Tip Typing long references at the command line invites both carpal tunnel syndrome and boredom. As an alternative, you can add a reference to the list of default references in the global CSC.rsp file, located at <WinDir>\Microsoft.NET\Framework\<Version>\CSC.rsp. For example, if you add */r: "c:\program files\common files\microsoft shared\msenv\publicassemblies\extensibility.dll"* to the global CSC.rsp file, you can compile the Basic add-in with the following command:

```
csc /t:library basic.cs
```

Registering the Basic Add-In with Visual Studio

Basic.dll is a fully functional add-in, but Visual Studio won't have the information that Basic.dll is in existence by this point. Add-ins signal their availability to the Visual Studio IDE through an XML file with an extension of .addin. Listing 6-3 shows the minimal .addin file for the Basic add-in.

Listing 6-3 The Basic.addin XML file

```
<?xml version="1.0" ?>
<Extensibility
  xmlns="http://schemas.microsoft.com/AutomationExtensibility">
    <HostApplication>
        <Name>Microsoft Visual Studio</Name>
        <Version>8.0</Version>
    </HostApplication>
    <Addin>
        <Assembly>basic.dll</Assembly>
        <FullClassName>Basic</FullClassName>
    </Addin>
</Extensibility>
```

As you can see from Listing 6-3, the Basic.addin file has a simple structure that describes the add-in and its host environment. The <HostApplication> node identifies the Visual Studio IDE as the add-in host; the <Addin> node gives the path to the add-in assembly (relative to the .addin file) and also provides the fully qualified name of the class that implements *IDTExtensibility2*.

To register the Basic add-in, either copy the Basic.addin and Basic.dll files to a location that's well known to Visual Studio, or point Visual Studio to the location of the two files. To accomplish the latter, run Visual Studio and choose Tools | Options to display the Options dialog box. From there, select the Environment\Add-in/Macros Security node in the tree view, click the Add button, and then select the folder that contains Basic.addin and Basic.dll. Click OK to add the new folder to the list of locations that Visual Studio will load add-ins from, and then click OK to close the Options dialog box.

After you register the Basic add-in, choose Tools | Add-in Manager to display the dialog box shown in Figure 6-5. Notice that the Add-in Manager lists the add-in's name as Basic. By default, the Add-in Manager displays the class name it finds under the <FullClassName> node in the .addin file.

If you select the Basic check box and click OK, Visual Studio loads the add-in. Assuming all goes well, Visual Studio will call Basic's *OnConnection* method, and you'll see the message box shown in Figure 6-6.

And that's how you create an add-in from scratch. In the next section, we'll examine exactly what happens to an add-in in the Visual Studio environment when the add-in loads, when the add-in unloads, and all the time in between.

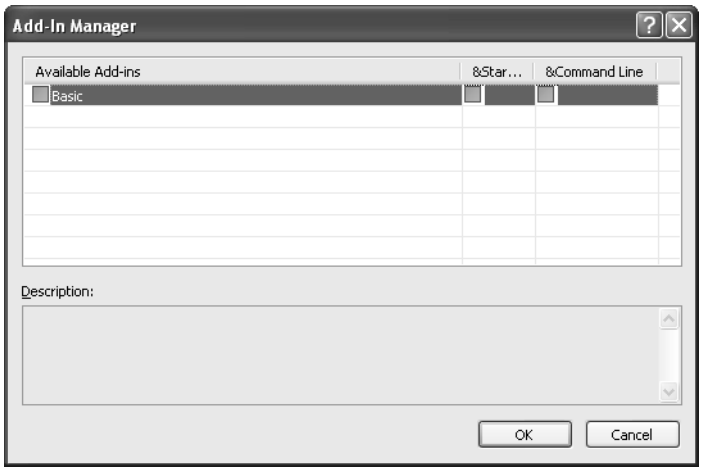


Figure 6-5 The Add-in Manager dialog box showing the Basic add-in

Figure 6-6 The message box displayed by the Basic add-in

Add-In Events

Add-ins are event-driven. Most everything an add-in does it does in response to some external prodding, and Visual Studio prods add-ins with the *IDTExtensibility2* interface. We'll begin our exploration of add-in events by examining the sequence in which Visual Studio calls the *IDTExtensibility2* methods.

The Add-In Event Sequence

Calls to the *IDTExtensibility2* methods, which we'll also refer to as *events*, occur at predictable points in the lifetime of an add-in. Figure 6-7 shows the sequence of events from the time an add-in is loaded to the time it is unloaded.

You can guess the actions that trigger the events just from the events' names, and the events occur pretty much in the order you would expect: *OnConnection* when an add-in loads, *OnDisconnection* when an add-in unloads, and so on.

You can get a feel for the add-in event sequence by running the LifeCycle sample add-in. LifeCycle, shown in Listing 6-4, handles each *IDTExtensibility2* event by displaying the name of the event in the Output window. After you build the LifeCycle add-in, load it into Visual Studio by using the Add-in Manager. Then try to load and unload other add-ins, such as Basic, to trigger the different *IDTExtensibility2* events. To fire the *OnStartupComplete* event,

you first need to select the Startup check box for LifeCycle in the Add-in Manager, and then you must restart Visual Studio. To fire the *OnBeginShutdown* event, close Visual Studio while LifeCycle is loaded.

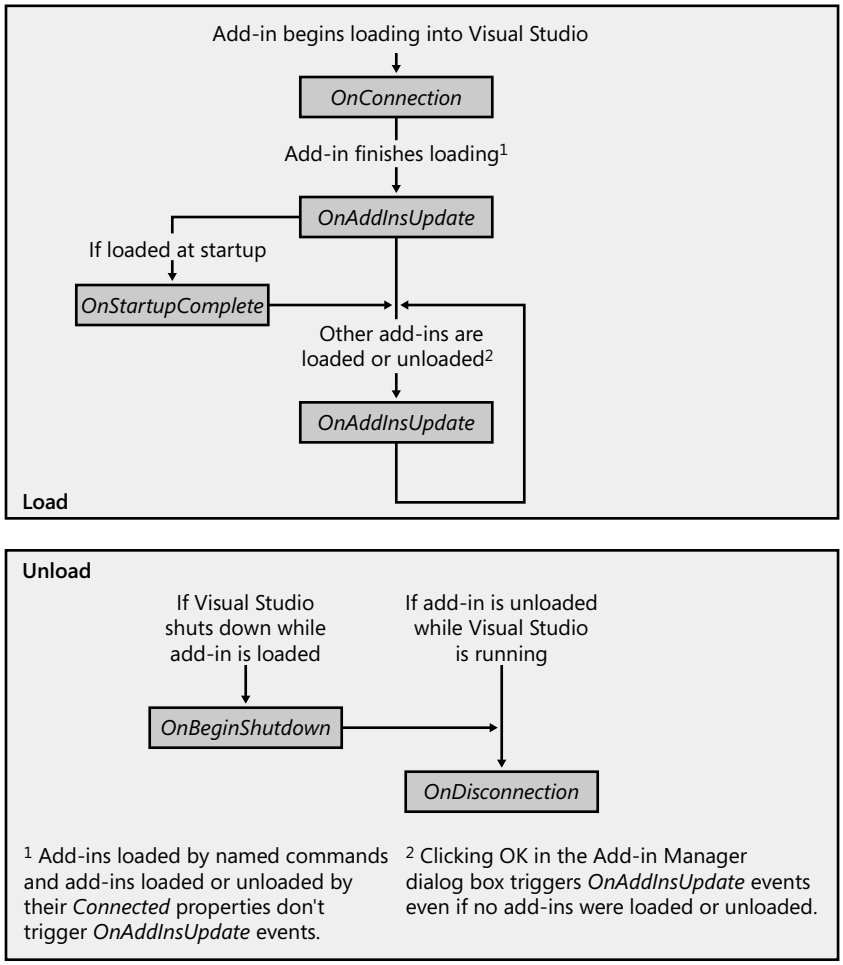


Figure 6-7 The add-in event sequence

Listing 6-4 LifeCycle.cs, the LifeCycle add-in source code

```
using EnvDTE;
using EnvDTE80;
using Extensibility;
using System;

public class LifeCycle : IDTExtensibility2
{
    private OutputWindowPane output;
```

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

        try
        {
            this.output =
               dte.ToolWindows.OutputWindow.OutputWindowPanes.Item(
               "LifeCycle");
        }
        catch
        {
            this.output =
                dte.ToolWindows.OutputWindow.OutputWindowPanes.Add(
                "LifeCycle");
        }

        this.output.OutputString("OnConnection event fired\n");
    }

    public void OnStartupComplete(ref Array custom)
    {
        this.output.OutputString("OnStartupComplete event fired\n");
    }

    public void OnAddInsUpdate(ref Array custom)
    {
        this.output.OutputString("OnAddInsUpdate event fired\n");
    }

    public void OnBeginShutdown(ref Array custom)
    {
        this.output.OutputString("OnBeginShutdown event fired\n");
    }

    public void OnDisconnection(ext_DisconnectMode removeMode,
        ref Array custom)
    {
        this.output.OutputString("OnDisconnection event fired\n");
    }
}
```

The *IDTExtensibility2* Interface

As you now know, an implementation of *IDTExtensibility2* lies at the core of every add-in. Visual Studio calls the methods on this interface whenever it needs to apprise an add-in of important events, such as when another add-in is loaded or unloaded or when Visual Studio is about to shut down. The communication isn't just one way, either: through the *IDTExtensibility2* interface, the add-in has access to and control over the entire Visual Studio automation object model.

The *EnvDTE* and *EnvDTE80* Namespaces

Before examining the individual *IDTExtensibility2* methods, we need to take a quick look at the real objective of add-ins—controlling the objects in the *EnvDTE* and *EnvDTE80* namespaces. The name *EnvDTE* stands for *Environment Development Tools Extensibility*, which pretty much describes its purpose: it defines the Visual Studio automation object model. (As you might guess, *EnvDTE80* defines the *8.0* version objects in the automation object model.) The Visual Studio documentation includes a chart of the automation object model that displays a hierarchy of more than 170 objects defined by the *EnvDTE* and *EnvDTE80* namespaces. The add-ins in this book will make use of most of those objects, but a few of the objects are of special interest to add-ins:

- **DTE/DTE2** The root objects of the automation object model
- **AddIn** An object that represents an add-in
- **DTE.AddIns/DTE2.AddIns** A collection of *AddIn* objects that includes all add-ins registered with the Visual Studio IDE
- **DTE.Solution.AddIns/DTE2.Solution.AddIns** A collection of *AddIn* objects associated with a solution

The next several examples will focus on the *DTE/DTE2*, *AddIn*, and *DTE.AddIns/DTE2.AddIns* objects, which collectively give you control over your own add-in and others. We'll cover the *DTE.Solution.AddIns* object in Chapter 9.

Note The main purpose of an add-in class is to provide an implementation of *IDTExtensibility2*, but that doesn't have to be its only purpose. An add-in class is a class, after all, and it can define any number of non-*IDTExtensibility2*-related methods, properties, and events. The automation object model provides access to your add-in class through the *AddIn.Object* property, which returns the add-in's *IDispatch* interface. The following macro code shows how you would call a public method named *DisplayMessage* on the *MyAddIn.Connect* add-in class:

```
Dim dispObj As Object = DTE.AddIns.Item("MyAddIn.Connect").Object
dispObj.DisplayMessage("IDispatch a message to you.")
```

OnConnection

OnConnection provides an add-in with the main object reference it needs to communicate directly with the IDE. The *OnConnection* method has the following prototype:

```
public void OnConnection(object application,
    ext_ConnectMode connectMode,
    object addInInst,
    ref Array custom);
```

The *application* parameter holds a reference to the root object of the automation object model. Technically, *application* holds a reference to an object that implements both the

EnvDTE.DTE and *EnvDTE80.DTE2* interfaces, so you can cast *application* to *EnvDTE.DTE* or *EnvDTE80.DTE2*, according to your needs. Almost every add-in that does something useful has need of the *DTE* or *DTE2* object, so the first statements in *OnConnection* typically cache the object in a global variable.

The *connectMode* parameter tells an add-in the circumstance under which it was loaded. This parameter takes on one of the *Extensibility.ext_ConnectMode* enumeration values shown in Table 6-2.

Table 6-2 The *Extensibility.ext_ConnectMode* Enumeration

Constant	Value (Int32)	Description
ext_cm_AfterStartup	*0x00000000*	Loaded after Visual Studio started
ext_cm_Startup	*0x00000001*	Loaded when Visual Studio started
ext_cm_External	*0x00000002*	Loaded by an external client (no longer used by Visual Studio)
ext_cm_CommandLine	*0x00000003*	Loaded from the command line
ext_cm_Solution	*0x00000004*	Loaded with a solution
ext_cm_UISetup	*0x00000005*	Loaded for user interface setup

An add-in can check the *connectMode* value and alter its behavior accordingly. For example, when an add-in receives the *ext_cm_UISetup* value, it can add its custom commands to the IDE menus and toolbars. (The Add-in Wizard generates code that handles the *ext_cm_UISetup* case in this manner.)

The *addInInst* parameter passes an add-in a reference to its own *AddIn* instance, which it can store for later use. (The *AddIn* instance proves invaluable for discovering the add-in's parent collection.) Finally, each of the *IDTExtensibility2* methods includes a *custom* parameter, which allows add-in hosts to pass in an array of host-specific data. Visual Studio always passes an empty array in *custom*.

OnStartupComplete

The *OnStartupComplete* event fires only in add-ins that load when Visual Studio starts. The *OnStartupComplete* prototype looks like this:

```
public void OnStartupComplete(ref Array custom);
```

An add-in that loads at startup can't always rely on *OnConnection* for its initialization—if the add-in arrives too early, it will fail when it tries to access a Visual Studio component that hasn't yet loaded. In such cases, the add-in can use *OnStartupComplete* to guarantee that Visual Studio is up and running first.

OnAddInsUpdate

The *OnAddInsUpdate* event fires when an add-in joins or leaves the Visual Studio environment. An add-in can use this event to enforce dependencies on other add-ins. Here's the *OnAddInsUpdate* prototype:

```
public void OnAddInsUpdate(ref Array custom);
```

The *OnAddInsUpdate* event doesn't provide you with information about which add-in triggered the event or why. If you need to know the add-in responsible for the event, you have to discover its identity on your own. Fortunately, you have the *DTE.AddIns/DTE2.AddIns* collection to aid you in your investigation. This collection holds a list of *AddIn* objects (one for each registered add-in), and each *AddIn* object has a *Connected* property that exposes its connection status. You retrieve a specific add-in from the *AddIns* collection by passing the *AddIns.Item* method a fully qualified class name or a 1-based index; if the requested index doesn't exist in the collection, the *Item* method throws an invalid index *COMException*; otherwise, it returns an *AddIn* reference. Here's one way to check LifeCycle's connection status:

```
public void OnAddInsUpdate(ref Array custom)
{
    try
    {
        AddIn addIn = this.dte.AddIns.Item("LifeCycle");

        if (addIn.Connected)
        {
            // LifeCycle is connected
        }
        else
        {
            // LifeCycle isn't connected
        }
    }
    catch (COMException)
    {
        // LifeCycle isn't a registered add-in
    }
}
```

Of course, whether LifeCycle caused the event remains a mystery. The LoadUnload add-in, shown in Listing 6-5, does what the previous sample cannot: it deduces which add-in triggers the *OnAddInsUpdate* event.

Listing 6-5 LoadUnload.cs, the LoadUnload source code

```
namespace LoadUnload
{
    using EnvDTE;
    using EnvDTE80;
    using Extensibility;
```

```
using Microsoft.VisualStudio.CommandBars;
using System;
using System.Collections.Generic;

public class Connect : Object, IDTExtensibility2, IDTCommandTarget
{
    public Connect()
    {
    }

    public void OnConnection(object application,
        ext_ConnectMode connectMode, object addInInst,
        ref Array custom)
    {
        this.applicationObject = (DTE2)application;
        this.addInInstance = (AddIn)addInInst;
        this.addInsCollection = this.applicationObject.AddIns;
        this.addInsList = new SortedList<string, bool>();

        foreach (AddIn addIn in this.addInsCollection)
            this.addInsList[addIn.ProgID] = addIn.Connected;

        OutputWindow win =
            this.applicationObject.ToolWindows.OutputWindow;

        try
        {
            this.output = win.OutputWindowPanes.Item("LoadUnload");
        }
        catch
        {
            this.output = win.OutputWindowPanes.Add("LoadUnload");
        }
            .
            .
            .
    }

    public void OnAddInsUpdate(ref Array custom)
    {
        this.addInsCollection.Update();

        foreach (AddIn addIn in this.addInsCollection)
        {
            string action = addIn.ProgID + " was ";

            if (this.addInsList.ContainsKey(addIn.ProgID))
            {
                if (addIn.Connected != this.addInsList[addIn.ProgID])
                {
                    action += addIn.Connected ? "loaded" : "unloaded";

                    this.output.OutputString(action + "\n");
                }
            }
```

```
                else
                {
                    action += "added" +
                        (addIn.Connected ? " and loaded" : "");

                    this.output.OutputString(action + "\n");
                }

                this.addInsList[addIn.ProgID] = addIn.Connected;
            }
        }
            .
            .
            .
        private DTE2 applicationObject;
        private AddIn addInInstance;
        private AddIns addInsCollection;
        private SortedList<string, bool> addInsList;
        private OutputWindowPane output;
    }
}
```

LoadUnload maintains a running list of add-ins and their connection statuses in its *addInsList* variable, which is declared as type *SortedList<string, bool>*. When *OnAddInsUpdate* fires, LoadUnload compares the connection statuses of the add-ins in its internal list with the connection statuses of the add-ins in the *DTE2.AddIns* collection—if it finds a discrepancy, it knows which add-in to blame for the event. Here's the first part of the main loop from Listing 6-5:

```
this.addInsCollection.Update();

foreach (AddIn addIn in this.addInsCollection)
{
    string action = addIn.ProgID + " was ";

    if (this.addInsList.ContainsKey(addIn.ProgID))
    {
        if (addIn.Connected != this.addInsList[addIn.ProgID])
        {
            action += addIn.Connected ? "loaded" : "unloaded";

            this.output.OutputString(action + "\n");
        }
    }
  §
```

The *addInsCollection* variable holds a reference to the *DTE2.AddIns* collection, and the call to *Update* synchronizes the collection with the registry so that any newly created add-ins are included. (The Add-in Manager performs the equivalent of *Update* each time it runs.) After the call to *Update*, the main loop iterates through the current add-ins in *addInsCollection* and checks whether each add-in already exists in its internal list. If so, the *Connected* property

of the add-in is compared with the corresponding value stored in the internal list; if they differ, the *Connected* property determines whether the add-in was loaded (true) or unloaded (false).

If the current add-in doesn't exist in *addInsList*, the add-in was registered some time between the previous *OnAddInsUpdate* event and this *OnAddInsUpdate* event. Here's the second part of the main loop, which handles new add-ins:

```
  §
    else
    {
        action += "added" +
            (addIn.Connected ? " and loaded" : "");

        this.output.OutputString(action + "\n");
    }

    this.addInsList[addIn.ProgID] = addIn.Connected;
}
```

The last statement either writes the current *Connected* value to an existing entry or creates a fresh entry for a newly registered add-in.

LoadUnload isn't foolproof—for example, add-ins loaded by commands arrive and leave unannounced—but it works well enough for demonstration purposes.

OnBeginShutdown

Here's the prototype for *OnBeginShutdown*:

```
public void OnBeginShutdown(ref Array custom);
```

This event fires only when the IDE shuts down while an add-in is running. Although an IDE shutdown might get canceled along the way, *OnBeginShutdown* doesn't provide a cancellation mechanism, so an add-in should assume that shutdown is inevitable and perform its cleanup routines accordingly. An add-in that manipulates IDE state might use this event to restore the original IDE settings.

OnDisconnection

This event is similar to *OnBeginShutdown* in that it signals the end of an add-in's life; it differs from *OnBeginShutdown* in that the IDE isn't necessarily about to shut down. *OnDisconnection* also provides more information to an add-in than *OnBeginShutdown* does. *OnDisconnection*'s prototype looks like this:

```
public void OnDisconnection(ext_DisconnectMode removeMode,
    ref Array custom);
```

The *removeMode* parameter passes in an *IDTExtensibility2.ext_Disconnect Mode* enumeration value that tells an add-in why it was unloaded. Table 6-3 lists the *ext_DisconnectMode* values.

Table 6-3 The *Extensibility.ext_DisconnectMode* Enumeration

Constant	Value (Int32)	Description
ext_dm_HostShutdown	*0x00000000*	Unloaded when Visual Studio shuts down
ext_dm_UserClosed	*0x00000001*	Unloaded while Visual Studio is running
ext_dm_UISetupComplete	*0x00000002*	Unloaded after user interface setup
ext_dm_SolutionClosed	*0x00000003*	Unloaded when solution closed

The *ext_DisconnectMode* enumeration serves a purpose similar to *ext_ConnectMode*: it allows an add-in to alter its behavior to suit its current circumstances. For example, an add-in that receives *ext_dm_UISetupComplete* probably would bypass its cleanup routines because it was loaded for initialization purposes only.

The .Addin File

As you learned earlier in this chapter, an add-in makes itself known to Visual Studio by registering an XML file with an .addin extension. The following sections explore the structure of this file in greater detail.

Host Application Information

The .addin file can specify any number of host applications by including a <*HostApplication*> element for each supported host. The <*HostApplication*> element contains two child elements: <*Name*> and <*Version*>, in that order. For Visual Studio add-ins, the <*Name*> value is either *Microsoft Visual Studio* to specify the main IDE as the host or *Microsoft Visual Studio Macros* to specify the Macros IDE as the host. The <*Version*> value identifies the version of Visual Studio that supports this add-in, such as *7.1* or *8.0*, or you can use a wildcard value of * to indicate that any version of Visual Studio will work.

The following (slightly contrived) example shows how to specify that a particular add-in works in all versions of the Macros IDE but only in the two most recent versions of the Visual Studio IDE:

```
<HostApplication>
    <Name>Microsoft Visual Studio Macros</Name>
    <Version>*</Version>
</HostApplication>
<HostApplication>
    <Name>Microsoft Visual Studio</Name>
    <Version>7.1</Version>
</HostApplication>
<HostApplication>
    <Name>Microsoft Visual Studio</Name>
    <Version>8.0</Version>
</HostApplication>
```

Practically speaking, specifying a version earlier than 8.0 makes little sense because previous versions of Visual Studio don't use .addin files for add-in registration.

Add-In Information

The next element in the .addin file—the <*Addin*> element—specifies the add-in itself, and the children of this element allow you to fine-tune the add-in's behavior. In order, the <*Addin*> element has the following child elements:

- <*FriendlyName*> (optional)
- <*Description*> (optional)
- <*AboutBoxDetails*> (optional)
- <*AboutIconData*> (optional)
- <*Assembly*> (required)
- <*FullClassName*> (required)
- <*LoadBehavior*> (optional)
- <*CommandPreload*> (optional)
- <*CommandLineSafe*> (optional)

The following several sections cover these elements and describe the effects that their values produce.

<FriendlyName> and <Description> The <*FriendlyName*> and <*Description*> elements allow you to apply a meaningful name and a short description to your add-in. An example of an application that uses these values is the Add-in Manager, which populates its add-in list with <*FriendlyName*> values and displays the <*Description*> value of the selected add-in in its Description box.

<*FriendlyName*> and <*Description*> each stores either a human-readable string or the ID of a string resource in the add-in's satellite DLL (in the form @<*resource ID*>).

<AboutBoxDetails> and <AboutIconData> The <*AboutBoxDetails*> and <*AboutIconData*> elements create space for your add-in on the Visual Studio About dialog box, as shown in Figure 6-8. The About dialog box displays <*FriendlyName*> values in the Installed Products list; when a user selects an add-in from this list, the dialog box displays both the icon in the <*AboutIconData*> element and the information in the <*AboutBoxDetails*> element in the Product Details list.

The format of the <*AboutBoxDetails*> value is the same as that of the <*FriendlyName*> and <*Description*> values—a string that holds either a short description or the ID of a string resource in the add-in's satellite DLL (in the form @<*resource ID*>). The <*AboutIconData*> value stores the binary data of an icon as an array of hexadecimal characters.

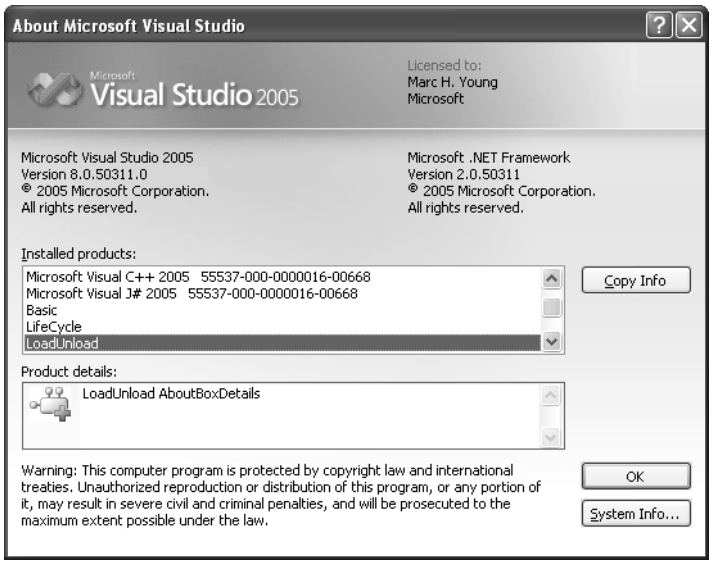


Figure 6-8 Add-in information displayed in the Visual Studio About dialog box

<LoadBehavior> The *<LoadBehavior>* value controls how an add-in is loaded by Visual Studio. Table 6-4 lists the possible *<LoadBehavior>* values. These values are bit flags, so you can combine them to create your own custom load settings.

Table 6-4 *<LoadBehavior>* Values

Value	Description
0x0	Add-in currently is unloaded.
0x1	Add-in loads at startup.
0x2	Add-in currently is loaded.
0x4	Add-in loads during command-line builds.

The *0x0* and *0x2* values no longer serve a useful purpose. Use the *AddIn.Connected* property instead to discover the load state of an add-in.

A value of *0x1* tells Visual Studio to load the add-in when the IDE starts up. Add-ins that monitor IDE events, in particular, need to be up and running from the beginning—otherwise, they might miss some of the action. Add-ins that don't monitor IDE events can omit this flag and wait to be loaded on demand.

A value of *0x4* signals that an add-in should be loaded during command-line builds.

<CommandPreload> Many add-ins expose their functionality through menu items and toolbar buttons in the IDE—when selected or clicked, these user interface items load the add-in and pass along the appropriate command for processing. Of course, the user interface items don't appear magically; an add-in creates them and adds them to the IDE the first time

that the add-in loads. But without user intervention, how does an add-in first get loaded in order to create the user interface item to load it? The solution to this problem begins with the <*CommandPreload*> value and ends with the PreloadAddinStateManaged registry key. An add-in sets its <*CommandPreload*> value to *0x1* to tell Visual Studio that it wants to be loaded once, the next time the IDE starts up, for the purpose of adding its user interface items to the IDE (a process known as preloading).

Note Actually, there is no requirement for preloaded add-ins to create user interface items; they're free to perform any kind of initialization they need, such as creating data files and adding custom registry entries.

At startup, Visual Studio examines every .addin file and preloads each add-in that has a <*CommandPreload*> value of *0x1*, but only if the add-in hasn't yet been preloaded. Visual Studio determines that an add-in hasn't been preloaded by searching the HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\8.0\PreloadAddinStateManaged registry key, which holds a list of already preloaded add-ins. If the PreloadAddinStateManaged key is missing or the add-in isn't on the list, Visual Studio has the information that the add-in hasn't yet been preloaded.

For each add-in that needs to be preloaded, Visual Studio loads the add-in, passing it the *ext_cm_UISetup* value in its *OnConnection* event, and then unloads it immediately after *OnConnection* returns, passing it the *ext_dm_UISetupComplete* value in its *OnDisconnection* event. After preloading an add-in, Visual Studio creates the PreloadAddinStateManaged key, if necessary, and sets the add-in's value to *0x2*.

<CommandLineSafe> The optional <*CommandLineSafe*> named value is supposed to work closely with the <*LoadBehavior*> value to ensure the success of unattended builds. A <*CommandLineSafe*> value of *0x1* indicates that the add-in won't display a user interface that requires human interaction—at least not when a build is started from the command line. A missing <*CommandLineSafe*> element or a <*CommandLineSafe*> value of *0x0* marks the add-in as unsuited for command-line builds. Currently, the <*CommandLineSafe*> value doesn't affect whether Visual Studio loads the add-in—the value is for informational purposes only.

Satellite DLLs

If you want to distribute your add-in internationally, you need to pay attention to the problem of localization. A worldly add-in doesn't force a particular language on its users—instead, it communicates with each user in his or her native tongue. Of course, a standalone add-in can't possibly accommodate every language; instead, an add-in that supports localization stores its localizable resources in satellite DLLs. At run time, the add-in searches for the satellite DLL that corresponds to the current locale and uses the localized resources from that DLL to populate its user interface. In this way, the same add-in can support any number of languages simply by providing a localized satellite DLL for each locale.

Each localized satellite DLL exists in its own folder under the add-in's root folder, and the name of the folder is the culture name that the DLL supports. For example, the culture name for U.S. English is en-US, so the U.S. English satellite DLL is found at *<add-in install path>*\en-US*<add-in name>*.resources.dll. At run time, you can locate an add-in's satellite DLL for the current culture by using either the *AddIn.SatelliteDllPath* property or the *DTE.SatelliteDllPath* method. (For more information about creating and using satellite DLLs, see the "Walkthrough: Creating Managed Satellite DLLs" and "How To: Access Resources in Satellite DLLs" Help topics.)

Looking Ahead

The next chapter is for that little bit of drill instructor inside all of us—the part that wants to hear others shout "HOW HIGH?" when we say "JUMP!" As you'll soon learn, macros, add-ins, and Visual Studio make the most loyal and obedient of soldiers, always at the ready and eager to do your bidding: all you have to do is give the right commands. If you turn the page, we'll show you how..."NOW!"

Chapter 7

Exploring Commands Programmatically

In this chapter:

Commands are the most fundamental mechanism of communication between the user and the Microsoft® Visual Studio® integrated development environment (IDE). In this chapter, we'll explore how you can use existing commands as well as create your own commands by using add-ins and macros.

What Is a Command?

If you've written user interface software for the Microsoft Windows® operating system, you're probably familiar with the event-driven programming model. When the user clicks a button on a form, chooses a menu item, or presses a key on the keyboard, your program receives a notification of that user action. If you're programming at the Windows SDK level, such as with the Microsoft Visual C++® programming language, when the user performs this action, your program receives a message detailing what happened. If you're using a language such as Microsoft Visual Basic® or Microsoft Visual C#®, this notification happens in the form of an event handler being called. These notifications are commands issued by the user, and the program carries out this command by performing some action for the user.

Visual Studio uses a method of notification similar to that of Win32® message-passing to inform code as the user interacts with the IDE. However, because of the complexity and number of commands available in the Visual Studio IDE, command *routing*, or passing a notification to the proper handler of that notification, isn't as simple as receiving a message. For instance, suppose the user chooses File | New | File. Because there are a number of different add-on programs (not to be confused with add-ins), such as Visual C++, Visual Basic, and Visual C#, Visual Studio needs to determine which of these programs handles this menu item choice. When a Win32 program handles a message, one message loop

handles that message, but because there are a number of possible handlers of a command in Visual Studio, commands need to be routed to the correct code. Each of these add-on products reserves a globally unique identifier (GUID) to uniquely identify itself, and each command that is available associates itself with the GUID of a particular add-on. When a user executes a command, the GUID for that command is retrieved, the add-on program that handles that GUID is found, and the command is sent to that add-on.

A command also needs another part to identify itself. After all, if every command had just a GUID to identify it, and all the commands that belonged to an add-on had the same GUID, an add-on wouldn't be able to tell the difference between, for instance, the New File command and the New Project command. To disambiguate commands that all have the same GUID, a number, or ID, is assigned to each command in that group. An add-on is responsible for its own commands, so an ID can be assigned without conflicting with commands from a different add-on because the GUID for each add-on is different. When combined, this GUID and ID pair uniquely identifies each individual command.

Note A command in Visual Studio exists independently of any user interface elements (such as menu items) for that command. Commands can be created and destroyed, and a user interface element might have never been created for that command. But the opposite won't happen—a user interface element can't be created without having a corresponding command.

Locating Commands

In Visual Studio, all the commands that a user can issue are represented in the object model by a *Command* object, and the *Commands* collection contains a list of these objects. As do other collection objects, *Command* objects allow the use of standard enumeration constructs such as the keywords *foreach* in Visual C# or *For Each* in Visual Basic. Using these keywords, we can create a macro to walk the list of all *Command* objects:

```
Sub WalkCommands()
    Dim cmd As EnvDTE.Command
    For Each cmd In DTE.Commands
      'use the EnvDTE.Command object here
    Next
End Sub
```

The *Command* collection's *Item* method works a bit differently from the *Item* methods of other collection objects. *Commands.Item* accepts as a parameter the familiar numerical index, but it also accepts an additional optional argument. If you're using the numerical indexing method, you should set the second argument to *-1*. This method has an additional argument because, as mentioned earlier, a GUID and ID pair is used to uniquely identify a command. The GUID, in string format, is passed as the first argument, and the ID of the command is passed as the second argument when you're using the GUID and ID to index the *Commands*

collection. The following macro demonstrates finding the command for opening a file by using the GUID and ID pair:

```
Sub FindFileOpenCommand()
    Dim cmd As EnvDTE.Command
    cmd = DTE.Commands.Item("{5EFC7975-14BC-11CF-9B2B-00AA00573819}", 222)
End Sub
```

As you can see, code like this can be complicated to write because you need to find and learn the GUID and ID for every command (which would be hard to do because there can be thousands of them), and then you must type this pair correctly every time, which can be a source of programming errors. To help with finding a *Command* object, the *Commands.Item* method accepts another format for indexing the collection, which is easier to remember: the name of a command.

Command Names

Remembering the GUID and ID for every command can be a huge waste of brainpower, so Visual Studio defines an easier-to-remember textual representation for most commands. These names follow a general pattern: the text of the top-level menu on which the primary user interface element for the command is located followed by a period, the text of all submenus combined followed by a period, and finally the text of the menu item. Any non-alphanumeric characters (except for the period separators and underscores) are then removed from this string. So to use the earlier example of finding the *Command* object for the open file command and combine it with our newly found way of using a command name, a macro such as the following results:

```
Sub FindFileOpenCommandByName()
    Dim command As EnvDTE.Command
    command = DTE.Commands.Item("File.OpenFile")
End Sub
```

To find the GUID and ID pair of a command, you can use the *GUID* and *ID* properties of the *Command* object. We used these two properties to find the GUID and ID pair used in the *FindFileOpenCommand* macro shown earlier. This is the macro we used to find them:

```
Sub FindGuidIDPair()
    Dim guid As String
    Dim id As Integer
    Dim command As EnvDTE.Command
    command = DTE.Commands.Item("File.OpenFile")
    guid = command.Guid
    id = command.ID
    MsgBox(guid + ", " + id.ToString())
End Sub
```

You can see the entire list of all available commands from within the Options dialog box on the Environment | Keyboard page. You can also use the object model to find available

command names. We'll do this with the *EnvDTE.Commands* collection in this example macro:

```
Sub CreateCommandList()
    Dim command As EnvDTE.Command
    Dim output As New OutputWindowPaneEx(DTE, "Create Command List")

    For Each command In DTE.Commands
        If (command.Name <> Nothing) Then
            output.WriteLine(command.Name)
        End If
    Next
End Sub
```

When the macro is run, it places into the Output window the name of each command. If you examine the macro closely enough, you'll notice a special check to verify that the name of the command isn't set to *Nothing*. This check is done because if a command doesn't have a name set, it returns *Nothing* if it's using Visual Basic or *null* if it's using C#. Commands that don't have a name are usually used internally by Visual Studio for private communication, and the user generally shouldn't call them. We advise you not to use these commands because they can lead to unpredictable results.

Executing Commands

The purpose of a command is to provide a way for the user to direct Visual Studio to perform some action. Commands can be invoked in a number of ways, the most common of which is for the user to choose a menu item or click a toolbar button. But commands can also be run in other ways. For example, if you write a macro that conforms to the standard macro notation (it is defined as public, doesn't return a value, and takes no arguments unless the arguments are optional strings), the macros facility detects that macro and creates a command for it. Double-clicking that macro in the Macro Explorer window executes the command associated with that macro, which is handled by the Macros editor. A third way to run a command is to use the *DTE.ExecuteCommand* method. This method runs a command, given by name, as if the user had chosen the menu item for that command.

To run our *File.OpenFile* command by using the *ExecuteCommand* method, we would write code like this:

```
Sub RunFileOpenCommand()
    DTE.ExecuteCommand("File.OpenFile")
End Sub
```

When a call is made to the *ExecuteCommand* method, execution of the macro or add-in waits until the command finishes executing.

A final approach, which is useful for the power user, is to type the name of the command into the Command Window. The Command Window is a text-based window in which you type the names of commands; when the user presses the Enter key, the command is run.

The command name that you type into the Command Window is the same name that is returned from the *Command.Name* property, and it can be passed to the *ExecuteCommand* method.

Creating Macro Commands

As mentioned before, macros that follow a special format are automatically turned into commands, and these macro commands are given a named counterpart. The name of a macro command is calculated by combining the string *Macros*, the name of the macro project, the name of the module or class the macro is implemented in, and finally the name of the macro with each portion separated by a period. Using this format, the *TurnOnLineNumbers* command in the Samples macro project that is installed with Visual Studio takes on the name *Macros.Samples.Utilities.TurnOnLineNumbers*. You can enter this name in the Command Window or call it from another macro, like so:

```
Sub RunCommand()
    DTE.ExecuteCommand("Macros.Samples.Utilities.TurnOnLineNumbers")
End Sub
```

Creating an Add-In Command

Now that you know how commands are named, found, and run, it's time to see how you can create your own commands. As we saw earlier, when a command built into Visual Studio is invoked, the add-on program for that command is located because of the GUID assigned to the command, and it is asked to handle the command invocation. Likewise, commands that you create need a target that handles the command invocation. Commands can be dynamically created and removed, but creating them requires that an add-in be associated with the new command so Visual Studio can find and use that add-in as the target. The method to create a command, *AddNamedCommand2*, can be found on the *Commands2* collection object. Here is its signature:

```
public EnvDTE.Command AddNamedCommand2(EnvDTE.AddIn AddInInstance,
    string Name, string ButtonText, string Tooltip, bool MSOButton,
    int Bitmap = 0, ref object[] ContextUIGUIDs,
    int vsCommandStatusValue = 3, int CommandStyleFlags = 3,
    EnvDTE80.vsCommandControlType ControlType = 2)
```

and here are the arguments:

- **AddInInstance** The *AddIn* object that will act as the command invocation target.
- ***Name*** The name of the command. The name can contain only alphanumeric characters and the underscore character.
- ***ButtonText*** The text that is displayed on any user interface elements, such as buttons, for the command when placed on menus or command bars.

- ***ToolTip*** Descriptive text providing users with information about the command.
- ***MSOButton*** *True* if the bitmap to display on user interface elements for this command should use the predefined command bar button graphics. If *False*, the graphic for the button is retrieved from the satellite DLL that is associated with the add-in.
- ***Bitmap*** If the *MSOButton* argument is *True*, the value passed to the *MSOButton* parameter is the index of the predefined command bar button graphic. See the HTML page in the CommandUIBmps folder included with the book's sample files for a listing of available images. If *MSOButton* is *False*, the *MSOButton* parameter is the resource identifier of the bitmap picture in the satellite DLL.
- ***ContextUIGUIDs*** Visual Studio defines a list of GUIDs that, as Visual Studio enters and exits a particular state, such as entering and exiting debugging mode, it marks as active and inactive. Some of these GUIDs are listed in the *EnvDTE80.ContextGuids* class. When a particular GUID becomes active and that GUID is passed to *AddNamedCommand2* through this parameter, the command will become visible. For example, suppose you have a command that helps users debug their code. You could pass the context GUID *vsContextGuidDebugging*, and when the user starts debugging, your command will use the default state passed for the next parameter, *vsCommandStatusValue*. If you do not have any context GUIDs for your command, an empty array of type *System.Object* should be passed for this value, and the state passed to *vsCommandStatusValue* will always be applied.
- ***vsCommandStatusValue*** This is the default availability state of the button. If the add-in that handles the command invocation has not yet been loaded, rather than forcing the add-in to load to find how the command should be displayed, this argument provides a default availability state. This argument value is used in place of the value returned through the *StatusOption* argument of the *QueryStatus* method on the *IDTCommandTarget* interface, which we'll discuss later in this chapter.
- ***CommandStyleFlags*** User interface elements for the new command can show just a bitmap (as the items on the standard toolbar do), just the name of the text (as the items on the menu bar do), or both. This parameter controls the appearance of the user interface element and is a value from the *vsCommandStyle* enumeration.
- ***ControlType*** When the command is created, no user interface element—such as a menu item, combo box, or most recently used (MRU) list—is created for that command. But if you intend to create a UI element for the command, you need to declare which kind of element will be created. This declaration is done through this parameter.

When called, the *AddNamedCommand2* method adds an item to the internal list of commands maintained by Visual Studio. The full name of the command, which you can use in the Command Window or as an argument to the *ExecuteCommand* method, is constructed by taking the fully qualified name of the class implementing the add-in (in the form of *Namespace.Class*) and concatenating a period, followed by the value of the *Name*

parameter. So, for example, if the name you provide to the *AddNamedCommand2* method is *MyCommand* and the *Namespace.ClassName* of the add-in is *MyAddin.Connect*, the name of the command that's created is *MyAddin.Connect.MyCommand*.

All commands added with this method also have a GUID and ID pair assigned to them. The GUID that is used for all commands created with *AddNamedCommand2* is defined by the constant *EnvDTE.Constants.vsAddInCmdGroup*; the ID value starts at the index 1 for the first call to *AddNamedCommand2*, and depending on the value passed to the *ControlType* parameter, it is incremented by anywhere from 1 to 25 values every time the *AddNamedCommand2* method is called.

Handling a Command Invocation

With a newly created command, our code now needs to provide a way for Visual Studio to call back to the add-in to let it know when the command is invoked. Usually, when an add-in or macro wants to be informed when the user has performed an action, an event connection is made. But command handlers work a bit differently: rather than connecting to an event source, your add-in must implement a specific interface. The reason for not using events is simple. When an add-in command is invoked, if the add-in that handles that command hasn't been loaded, the code for the add-in is loaded into memory and run by calling the *OnConnection* and other appropriate *IDTExtensibility2* methods, just as if you were to go into the Add-in Manager dialog box and select the check box for that add-in. Because the add-in is demand-loaded (loaded when the command is run), code within that add-in could not have been run to connect to an event handler.

The interface to handle command invocations, named *IDTCommandTarget*, is modeled on the *IOleCommandTarget* interface of the Win32 SDK, but *IDTCommandTarget* has been changed to be easier to use with languages such as C# or Visual Basic. This is its signature:

```
public interface IDTCommandTarget
{
    public void Exec(string CmdName,
        EnvDTE.vsCommandExecOption ExecuteOption, ref object VariantIn,
        ref object VariantOut, ref bool Handled);

    public void QueryStatus(string CmdName,
        EnvDTE.vsCommandStatusTextWanted NeededText,
        ref EnvDTE.vsCommandStatus StatusOption,
        ref object CommandText);
}
```

When these two methods are called and which values are passed to them depend on the type of command that you are adding to Visual Studio. The simplest case is when you pass the value *vsCommandControlTypeButton* for the *ControlType* parameter for *AddNamedCommand2*, and we will first discuss these two methods in terms of this command type. When invoked, all commands that your add-in creates are dispatched

through this interface, particularly through the *Exec* method. The *Exec* method has the following arguments:

- ***CmdName*** The full name of the command. Your add-in should do a case-sensitive compare on this string to determine which command is being asked to run because all commands that the add-in creates are sent to this method for handling.
- ***vsCommandExecOption*** For most situations, the value passed to this parameter is the *vsCommandExecOptionDoDefault* enumeration value, informing your add-in that it should do the work defined for that command.
- ***VariantIn*** As you'll see later in this chapter, commands can be passed data. If any data is passed to your command, they are passed through this argument.
- ***VariantOut*** This argument is used to pass data from your add-in to the caller.
- ***Handled*** This argument allows your add-in to pass back data to Visual Studio, signaling whether your add-in handled the command. If a true value is returned, it is assumed that no further processing for the command is necessary. If this value is set to false on return, Visual Studio continues searching for a handler for the command. The search should fail because no other command handler will accept the same GUID and ID pair for the command your add-in has created.

Command State

A command and its user interface don't always need to be enabled and available to the user. For example, your add-in's command might be available only when a text editor is the currently active window. You can control whether your command is enabled, disabled, or in the *latched* state (which means a check mark is drawn next to the button if it is a menu item or appears with a box drawn around it if it is on a toolbar). You control this state by using the *QueryStatus* method of the *IDTCommandTarget* interface. If your add-in hasn't yet been loaded, the default status, or value passed as the last argument of *AddNamedCommand*, is used to control the default behavior. However, once you've loaded the add-in—by executing the command or manually through the Add-in Manager dialog box—*QueryStatus* is called to determine the state. The *QueryStatus* method has the following arguments:

- ***CmdName*** This argument has the same meaning as the *CmdName* argument passed to the *Exec* method of the *IDTCommandTarget* interface.
- ***NeededText*** This parameter is always *vsCommandStatusTextWantedNone*. Your add-in should always verify that this value is passed because the other values are reserved for future versions of Visual Studio.
- ***StatusOption*** Your add-in should fill in this parameter, which lets Visual Studio know whether the add-in command is supported (*vsCommandStatusSupported*) or unsupported (*vsCommandStatusUnsupported*), whether the command is enabled and can be called (*vsCommandStatusEnabled*), whether the command user interface can't be

seen (*vsCommandStatusInvisible*), or whether the user interface is drawn in the selected state (*vsCommandStatusLatched*). You can logically OR these values together to create the current status of the command and pass it back through this argument.

- ***CommandText*** This value currently isn't used by Visual Studio and shouldn't be modified.

Periodically, such as when the focus changes from one window to another or when a menu is displayed that contains an add-in command, Visual Studio calls *QueryStatus* for that command to ensure that the user interface is synchronized with the command state. It is important to keep the code that implements *QueryStatus* as efficient as possible; otherwise, the user interface might become sluggish. Suppose you create a command that queries the currently active file's attributes, and the state of a command depends on those file attributes. If the file is on the local disk, calling a method to retrieve the attributes of a file is a relatively fast operation. But if the file is on a network share, that operation can take awhile to perform—especially if the network is temporarily unavailable. A user who has to wait for a command to update itself because he or she showed the menu containing your command would be much happier if the command were always enabled and he or she would receive an error message when the command was invoked.

MRU Button Commands

If you were to choose File | Recent Files, you will see a list of files that you have recently opened. This list of files is implemented using a most recently used menu item list, or MRU button list, and is created with a command type of *vsCommandControlTypeMRUButton*. An MRU button list allows you to create many related commands with one call to the *AddNamedCommand2* method. When UI for these commands are added to a menu, multiple menu buttons are created and grouped together. When this command type is used, Visual Studio will create 25 separate commands rather than just the one it would create when the command type is *vsCommandControlTypeButton*. If the name of the command supplied to the *AddNamedCommand2* method is, for example, MRUButton, and the full name of the class implementing *IDTCommandTarget* is *MyAddin.Connect*, then *AddNamedCommand2* will create commands named *MyAddin.Connect.MRUButton*, *MyAddin.Connect.MRUButton_1*, *MyAddin.Connect.MRUButton_2*, and so on to *MyAddin.Connect.MRUButton_24*. This allows you to create an MRU list of 25 separate items.

For this command type, the *QueryStatus* method is used not only to retrieve the status of the command but also to retrieve the text of the menu item. If your add-in has only four separate MRU items, for the commands *MyAddin.Connect.MRUButton*, *MyAddin.Connect.MRUButton_1*, *MyAddin.Connect.MRUButton_2*, and *MyAddin.Connect.MRUButton_3*, your query status method returns as the value for status *vsCommandStatus.vsCommandStatusSupported*. For these values, the *commandStatus* will also not be *null/Nothing*, and it is your opportunity to set the text displayed for the menu item. MRU items can have text that changes frequently because (such as in the list of open files) the user will open, close, and remove files quite often, and you will want to keep that list as up-to-date as possible. Because, for this example,

you want to display only four MRU menu items, the *QueryStatus* method should return *vsCommandStatus.vsCommandStatusUnsupported* for *MyAddin.Connect.MRUButton_4* through *MyAddin.Connect.MRUButton_24*. This bit of code shows how to implement the *QueryStatus* method for an MRU button list. It maintains a list of four items, which is used for the text of the four menu items. When executed, the code will look at the command name, and if the name is out of the acceptable range, *vsCommandStatusUnsupported* is passed back to the caller. If the command name is in the appropriate range, it sets the *commandText* parameter to a string that will be displayed on the menu item for that MRU item, sets the status parameter to a value indicating the command is available, and then returns.

```
string []MRUItemNames = new string[]
  {"Item 1", "Item 2", "Item 3", "Item 4" };
void QueryStatus(string commandName,
    vsCommandStatusTextWanted neededText,
    ref vsCommandStatus status, ref object commandText)
{
if (neededText == vsCommandStatusTextWanted.vsCommandStatusTextWantedNone)
    {
        int index = 0;
        string rootCommandName = "MyAddin.Connect.MRUButton";
        if (commandName.StartsWith(rootCommandName))
        {
               //Check for the main command of the group
            if ("MyAddin.Connect.MRUButton" == commandName)
            {
                index = 0;
            }
            else //Command is of the form MyAddin.Connect.MRUButton_*
            {
                index = int.Parse(commandName.Substring(rootCommandName.Length + 1));
            }
            if (index > 3)
              //If the command index is out of the
              //range of MRU items supported...
            { //then the command is should not show
                status = vsCommandStatus.vsCommandStatusUnsupported;
                return;
            }
            else
            { //then show the command, and set the text of the item
                status = (vsCommandStatus)
                  vsCommandStatus.vsCommandStatusSupported |
                  vsCommandStatus.vsCommandStatusEnabled;
                commandText = MRUItemNames[index];
                return;
            }
        }
    }
}
```

Visual Studio may also call the *QueryStatus* method just to retrieve the text on the menu item and not require the status of the command. In this case, the *neededText* parameter will

be set to the value *vsCommandStatusTextWantedName*. You can change the aforementioned *QueryStatus* to also check for this value, and because both status and *commandText* are set (setting one value when it is not needed does not cause any side effects—Visual Studio just ignores it) in this code, you will handle both cases:

```
void QueryStatus(string commandName,
  vsCommandStatusTextWanted neededText,
  ref vsCommandStatus status, ref object commandText)
{
if ((neededText == vsCommandStatusTextWanted.vsCommandStatusTextWantedNone) ||
    (neededText == vsCommandStatusTextWanted.vsCommandStatusTextWantedName))
    {
        //TODO: Code to handle the QueryStatus
    }
}
```

The *Exec* method for this control type is very much like the *Exec* method for a *vsCommandTypeButton* command type, except that you will have multiple, similarly named items being passed to the *Exec* method. Code to find which item was executed will look similar to that in the *QueryStatus* method, except, rather than returning the text and status of the command, you will perform the appropriate action for that command:

```
void Exec(string commandName,
      vsCommandExecOption executeOption,
      ref object varIn, ref object varOut, ref bool handled)
{
    string rootCommandName = "MyAddin.Connect.MRUButton";
    handled = false;
    if (executeOption == vsCommandExecOption.vsCommandExecOptionDoDefault)
    {
        if (commandName == rootCommandName)
        {
            //Do the operation for MRUItemNames[0]
            handled = true;
            return;
        }
        else if (commandName.StartsWith(rootCommandName))
        {
            int index = int.Parse(commandName.Substring(rootCommandName.Length + 1));
            //Do the operation for MRUItemNames[index]
            handled = true;
            return;
        }
    }
}
```

Drop-Down Combo Boxes and MRU Combo Boxes

These command types are used to implement a combo box on a command bar. The difference between the two command types is in how the user interacts with them. An MRU combo box lets the user enter new text and also click the arrow to the right of the drop-down

combo box to select items that were entered in the past. This is similar to the find combo box on the standard command bar. A drop-down combo box is more restrictive; it allows the user to select only items that you choose from within your add-in code, and the user cannot enter new items. This type of combo box is similar to the Solution Configuration drop-down.

The items in a drop-down combo box are filled in through a call to the *Exec* method. When the command type is a *vsCommandControlTypeDropDownCombo*, the *AddNamedCommand2* method will create two separate commands. One command is used to retrieve the text that is shown within the main portion of the combo box, indicating the currently selected item. This command is also used to notify your code that the user has executed the command, meaning that the when the user selects an item from within the combo box, the *Exec* method is called with the text of the item that was selected. The second command that is created is used to retrieve the text of the items that are shown within the drop-down portion of the combo box, listing the possible items that the user can select. When the *AddNamedCommand2* method is called, the fully qualified name of the class implementing the add-in in the form of *Namespace.ClassName* is combined with the name of the command you supply. So if the class name is *MyAddin.Connect* and the command name is *DropDownCombo*, the resulting name of the first command that is created is *MyAddin.Connect.DropDownCombo*. The second command has this same name, except the text "_1" is appended, making the command name *MyAddin.Connect.DropDownCombo_1*. When the method *Exec* is called with a command name of *MyAddin.Connect.DropDownCombo*, you will need to check the values of the parameters *varIn* and *varOut* to determine if the list of items to fill in the drop-down portion is being asked for, or if a selection was made within the drop-down. If the value of *varIn* is not *null/Nothing*, and if the value it contains is a string, a selection was made within the drop-down. However, if the *varOut* value is not *null/Nothing*, your add-in is being queried for the currently selected item text to show in the drop-down, and you should set it to a string. Here is an example of the code implementing this first command:

```
void Exec(string commandName,
  vsCommandExecOption executeOption,
  ref object varIn, ref object varOut, ref bool handled)
{
    if (commandName == "MyAddin.Connect.DropDownCombo")
    {
        if ((varIn != null) && (varIn is string))
        {
            //The command was executed, retrive the selected text.
            string selectedText = varIn as string;
            //Perform some operation on the selected text.
        }
        else
        {
            //The selected text is being asked for, return that here.
            varOut = "Item 1";
        }
```

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

If the *Exec* method is called with the command name of the second command, the *varOut* value will not be *null*/*Nothing*, and you need to pass back an array containing strings of items to show. This code will fill in the drop-down portion with four items:

```
if (commandName == "MyAddin.Connect.DropDownCombo_1")
{
    //Set the list of items the user can select.
    varOut = new string[] { "Item 1", "Item 2", "Item 3", "Item 4" };

    handled = true;
    return;
}
```

The items in an MRU combo box are not filled in by calling the add-in; rather, the items are filled in by the user. The only code that you need to write to handle this command type, which is *vsCommandControlTypeMRUCombo*, is the *QueryStatus* code to indicate whether the command is enabled and the *Exec* code to handle selection within the drop-down. For the *Exec* method, the *varIn* parameter contains the text of the selected item when a selection is made. When Visual Studio closes, it will automatically save all the items that the user entered into the MRU combo box, and then it will restore them the next time Visual Studio is started.

Located within the samples that accompany this book, the add-in sample named CommandTypes demonstrates how to use each of these command types.

Programmatically Determining Command State

At times, you might need to programmatically determine whether a command is enabled and can be invoked, such as when you want to invoke a command by using *DTE.ExecuteCommand*. All commands, whether a macro command, one created by an add-in, or one built into Visual Studio, support a *QueryStatus* method. When you invoke the *DTE.ExecuteCommand* but the command isn't enabled because the *QueryStatus* method returned a value indicating that it isn't currently available, you'll get an exception if you're using a language supported by the .NET Framework.

To check whether a command is enabled and thus avoid this error condition, you can use the *Command.IsAvailable* property. For example, to make sure that the *Build.BuildSolution* command can be called before you invoke it, you can use the following code:

```
Sub CheckAvailability()
    If (DTE.Commands.Item("Build.BuildSolution").IsAvailable = True) Then
        DTE.ExecuteCommand("Build.BuildSolution")
    End If
End Sub
```

How an Add-In Command Handler Is Found

When a user invokes your command, Visual Studio needs to know which add-in handles that command so it can call the methods of the *IDTCommandTarget* interface. It first inspects the command name; as noted earlier, the first part of the full command name is the Namespace.ClassName of the add-in, and the remainder is the value passed for the *Name* parameter of the *AddNamedCommand* method. To locate the add-in, Visual Studio extracts the Namespace.ClassName from the command name and then checks the add-in corresponding to that Namespace.ClassName to see whether it's loaded. If it isn't, it is told to load. Visual Studio looks for the *IDTCommandTarget* interface (which must be implemented on the same object that implements *IDTExtensibility2*) on the add-in object instance, and then it calls the *Exec* method, passing the name of the command as the first parameter.

If, during this process, the add-in can't be found, the user is presented with the message box shown in Figure 7-1.

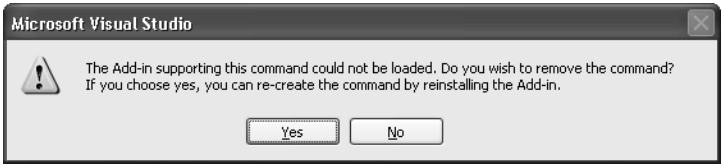


Figure 7-1 The message box displayed by Visual Studio when a command's add-in doesn't load

If the user clicks the Yes button, the command is removed using the *Command.Delete* method, and any user interface elements for that command are removed. If the add-in is loaded but the *IDTCommandTarget* interface can't be found on the add-in object, the command is treated as if the *QueryStatus* method had returned the *vsCommandStatusUnsupported* flag.

The Command User Interface

Visual Studio borrows its toolbar and menu system from the Microsoft Office suite of applications. The command bars provide a common user interface experience across all of the Office applications as well as Visual Studio. Because the command bars also support an object model, these applications also share a common programming model for accessing the command bar structure.

The main point of access to the command bar objects is through the *DTE.CommandBars* property. This property returns a *System.Object* type, which can be converted into a *Microsoft.VisualStudio.CommandBars.CommandBars* object, which is defined in the assembly Microsoft.VisualStudio.CommandBars.dll. The following macro code demonstrates retrieving this object:

```
Sub GetCommandBars()
    Dim commandBars As Microsoft.VisualStudio.CommandBars.CommandBars
    commandBars = DTE.CommandBars
End Sub
```

The Command Bar Object Model

The command bar object model is arranged in a treelike hierarchy, in the same way as the Visual Studio object model. At the top of this tree is a collection of *Microsoft.VisualStudio.CommandBars.CommandBar* objects that includes all the command bars, shortcut menus, and the main menu bar. Each command bar contains a collection of controls that have the type *Microsoft.VisualStudio.CommandBars.CommandBarControl*. Once a *CommandBarControl* is retrieved, it can be converted into one of a number of different types. One type, a *CommandBarButton*, is any menu item on a command bar that the user can click to perform an action; this is analogous to executing a Visual Studio command. To get to a *CommandBarButton* object, a cast must be performed from the *CommandBarControl* object:

```
Sub GetCommandBarButton()
    Dim commandBarBtn As Microsoft.VisualStudio.CommandBars.CommandBarButton
    Dim commandBarCtl As Microsoft.VisualStudio.CommandBars.CommandBarControl
    Dim commandBarCtls As Microsoft.VisualStudio.CommandBars.CommandBarControls

    'Find the View command bar
    commandBarCtls = DTE.CommandBars.Item("View").Controls
    'Retrieve the first control on the menu
    commandBarCtl = commandBarCtls.Item(1)
    'Convert the CommandBarControl to a CommandBarButton object
    commandBarBtn = CType(commandBarCtl, _
        Microsoft.VisualStudio.CommandBars.CommandBarButton)
    MsgBox(commandBarBtn.Caption)
End Sub
```

The object returned from the *Controls* collection can be converted into a *CommandBarPopup* if the item is the root node of a submenu. An example of this is the New item on the File menu; when the user holds the mouse cursor over this menu, a submenu appears. You can also retrieve a *CommandBarPopup* when the item is on a split-button drop-down menu, such as the New Project | New Blank Solution button on the Standard command bar:

```
Sub GetCommandBarPopup()
    Dim commandBar As Microsoft.VisualStudio.CommandBars.CommandBar
    Dim cmdBarControl As Microsoft.VisualStudio.CommandBars.CommandBarControl
    Dim cmdBarPopup As Microsoft.VisualStudio.CommandBars.CommandBarPopup

    'Find the "Standard" command bar
    commandBar = DTE.CommandBars.Item("Standard")
    'Find the first control on the command bar
    cmdBarControl = commandBar.Controls.Item(1)
    'Convert the CommandBarControl to a CommandBarPopup
    cmdBarPopup = CType(cmdBarControl, _
        Microsoft.VisualStudio.CommandBars.CommandBarPopup)
    MsgBox(cmdBarPopup.Controls.Item(1).Caption)
End Sub
```

A popup menu is itself a command bar. You can't cast directly to a *CommandBar* object on a popup menu, but this object does contain a *CommandBar* property, which returns a

CommandBar object, which itself has a collection of controls (as you can see in the next-to-last line in the preceding macro code).

The Primary Command Bar

The *DTE.CommandBars* property returns the collection of all *CommandBar* objects available within Visual Studio, but the most commonly used command bar is the main menu. Looking at the menu, you can see the File, Edit, and View items as well as a number of additional menu items; all of these are *CommandBar* objects within the *DTE.CommandBars* collection. But because there might be multiple items within the collection with the same name, indexing the collection by using the name might not work. For example, there are multiple *CommandBar* objects with the title View, and you might not always get the one you want if you index the *CommandBars* collection with the string *View*. The following macro might return the View command bar for the SQL editor, a deployment project popup menu, or the View menu:

```
Sub GetView()
    Dim cmdbars As Microsoft.VisualStudio.CommandBars.CommandBars
    Dim commandBar As Microsoft.VisualStudio.CommandBars.CommandBar

    cmdbars = DTE.CommandBars
    commandBar = cmdbars.Item("View")
End Sub
```

To work around this, you can find the *CommandBar* object for the menu bar, called *MenuBar*, and then find the View submenu command bar:

```
Sub GetMenuCommandBar()
    Dim commandBar As Microsoft.VisualStudio.CommandBars.CommandBar
    Dim cmdBarControl As Microsoft.VisualStudio.CommandBars.CommandBarControl
    Dim cmdBarPopupView As Microsoft.VisualStudio.CommandBars.CommandBarPopup
    Dim cmdBarView As Microsoft.VisualStudio.CommandBars.CommandBar

    'Retrieve the MenuBar command bar
    commandBar = DTE.CommandBars.Item("MenuBar")
    'Find the View menu
    cmdBarControl = commandBar.Controls.Item("View")
    'Convert to a CommandBarPopup
    cmdBarPopupView = CType(cmdBarControl, _
        Microsoft.VisualStudio.CommandBars.CommandBarPopup)
    'Get the CommandBar object for the view menu
    cmdBarView = cmdBarPopupView.CommandBar
    MsgBox(cmdBarView.Name)
End Sub
```

By default, if the Add-in Wizard generates an add-in and the option is selected to place an item on the Tools menu, code is generated to place a menu item on the Tools menu of the menu bar. If you want to move this command user interface to a different menu, you can simply change the string *Tools* to a different menu title, but be careful to select the correct menu. It's easy to make the mistake of selecting the wrong command bar, causing the

command button to seemingly disappear because it was placed somewhere that you did not expect it to go.

Adding New Command Bar Elements

With a *Command* object in hand (found by either indexing the *Commands* collection or adding a new command), and after using the methods described earlier to find the proper command bar, you can add new UI elements to that command bar that invokes your command when clicked. You do this by using the *Command.AddControl* method. When a control is added, the control type of the command (either passed to the call of *AddNamedCommand2* or the type of one of the built-in commands) is used to create the appropriate command type, be that a menu item, an MRU menu item list, or one of the two combo box types. When commands are created, they are persisted to disk and re-created automatically the next time Visual Studio is started. Likewise, when you place a control on a command bar by using the *AddControl* method, that control and its placement are saved to disk and re-created when Visual Studio is run. The first argument of the *AddControl* method is the *CommandBar* object that the button is to be placed on. The second argument defines the numerical position of the control in relation to the other controls on the command bar. (If this value is *1*, the control will be the first item on the command bar, and if the value is *2*, it will be the second item, and so forth.)

You can hard-code an index to place the control, but the control might not appear where you think it should go in relation to other controls. The reason is that a command bar might have one or more separators (or lines drawn between two controls) that divide controls into logical groups. These groups are also controls on the command bar, and they should be counted when you calculate the position. Not only are group controls counted as items in the index, but so are controls that are not visible because the value *vsCommandStatusInvisible* is returned from your *QueryStatus* method. If the control to be added should be placed at the bottom or end of the command bar, you can use the *Controls.Count* property to determine the final position:

```
Sub AddControl()
    Dim command As EnvDTE.Command
    Dim commandBar As Microsoft.VisualStudio.CommandBars.CommandBar

    'Find the File.OpenFile command
    command = DTE.Commands.Item("File.OpenFile")
    'Find the Tools CommandBar
    commandBar = DTE.CommandBars.Item("Tools")
    'Add a control to the Tools menu that when
    ' clicked will invoke the File.OpenFile command
    command.AddControl(commandBar, commandBar.Controls.Count + 1)
  End Sub
```

Note that the index used doesn't fix a control to a particular position. If you add a control to position 1 and a second control is added to position 1, the first control is pushed into the second position.

At times, it might make sense to create a new command bar to place your buttons on because the default set of command bars doesn't suit your needs. The command bar object model allows you to create new command bars, but creating one in this way might not achieve the desired effects. Command bars created in this way are created in a temporary state, which means that when you exit and restart Visual Studio, the command bar will have been destroyed. Because the button user interface for commands persists across instances, you'll want your command bars to also persist across instances. The Visual Studio object model lets you do this by using the *Commands.AddCommandBar* method, which has this signature:

```
object AddCommandBar(string Name, EnvDTE.vsCommandBarType Type, _
  Microsoft.VisualStudio.CommandBars.CommandBar
  CommandBarParent = null, int Position = 1)
```

This method has the following arguments:

- ***Name*** The caption to display on the command bar.
- ***Type*** A value from the *vsCommandBarType* enumeration. If the value is *vsCommandBarTypeToolbar*, a command bar is created that can be docked to the top, left, bottom, or right of the Visual Studio window. If the value is *vsCommandBarTypeMenu*, the command bar is added as a submenu to another command bar. If the value is *vsCommandBarTypePopup*, a shortcut menu is created.
- ***CommandBarParent*** If the value passed for the *Type* parameter is *vsCommandBarTypeToolbar* or *vsCommandBarTypePopup*, this value should be *null* or *Nothing* (depending on the language used). If the value passed to the *Type* parameter is *vsCommandBarTypeMenu*, the new menu should be rooted on the command bar object.
- ***Position*** This value is necessary only if the *Type* parameter is set to *vsCommandBarTypeMenu*. It defines the location on the parent command bar where the new menu command is placed. It has the same meaning as the *Position* parameter of the *AddControl* method.

How the newly created command bar is shown to the user depends on the type of command bar that's created. If the command bar type is a new menu, the menu item is hidden from the user until the command bar for that menu item is populated with buttons. If the command bar created is a new toolbar, the *Visible* property of the returned *CommandBar* object should be set to *True*. If a popup menu is created, you can show the menu to the user by using the *CommandBar.ShowPopup* method, which takes two arguments, the *x* and *y* coordinates of the top left of where the popup menu should appear.

Using Custom Bitmaps

Visual Studio has a number of predefined bitmaps that you can place on menu items and command bar buttons, but they might not always meet your needs. To use your own bitmap for the image on a button, you must create for your add-in a satellite DLL that contains

the bitmap as a resource, and then change the call to *Commands2.AddNamedCommand2* so Visual Studio can find your bitmap. First, you should set the *AddNamedCommand2* method's *MSOButton* parameter to *false* to tell Visual Studio that the bitmap isn't among the default, built-in pictures but is in the satellite DLL. Second, you should change the *Bitmap* parameter to the resource name of the bitmap in your satellite DLL; however, unlike other places where resource names can use alphanumeric values for the resource names, this method accepts only a number in the form of a string for the resource name. The bitmap must be in a specific format to be usable by Visual Studio. It must be 16 pixels high and 16 pixels wide, and it must be saved so that it has either 4-bit color (16 colors) or 32-bit color (4,294,967,296 colors). Visual Studio can also draw the picture so that a portion of it shows as transparent, causing the command bar background to bleed through. To enable this, you must make the transparent area have the *RGB* (red, green, blue) color value of *0, 254, 0*. Note that this color isn't the lime green color displayed in the color palette of the Visual Studio image editor or the Windows Paint application, and you will need to use the palette color manipulation features built into those tools.

Once you have created the .resx file and the bitmap within that file, you then need to generate the satellite DLL. I like to use the command line console rather than Visual Studio to generate the DLL. This is purely a matter of personal preference; you can use Visual Studio to build it for you. Using the integrated resource editor makes editing the .resx file much easier.

The sample CommandTypes, which was used earlier to demonstrate creating commands that implement combo-boxes and MRU menu item lists, also uses a custom bitmap with transparency for some of the command menu items. The call to *AddNamedCommand2* has been modified as described earlier; all that is left is to generate the satellite DLL. In the Localization folder, along with the CommandTypes sample is a batch file, MakeSatelliteDLL.bat, which will generate the satellite DLLs. This batch file repeats the following bit of code many times, but for a selection of different languages:

```
Resgen CommandTypes.en-US.resx
Al.exe /t:lib /embed:CommandTypes.en-US.resources
  /culture:en-US /out:CommandTypes.resources.dll
md .\..\CommandTypes\bin\en-US
copy CommandTypes.resources.dll .\..\CommandTypes\bin\en-US
```

The first line of this code takes a resource file, in this case the resources for the U.S. English culture, and creates a .resources file with the name CommandTypes-en-US.resources. Next, the assembly linker tool (Al.exe) is called to build a library DLL with the U.S. English resources embedded within it, and it names that file CommandTypes.resources.dll. The final two lines simply copy the file into a folder named en-US that is located in the same directory as the add-in DLL. When you call *AddNamedCommand2* from your add-in and the *MSOButton* parameter is set to *false*, Visual Studio will search for the satellite DLL assembly, load it, find the specified bitmap resource, and then apply it for the command UI that is created for that command. You do not need to modify the .addin file to specify the satellite

DLL; Visual Studio will find it by using the .NET Framework's *System.Reflection.Assembly.GetSatelliteAssembly* method.

Regenerating Commands and Their User Interface

As you are developing your add-in, you may need to add, remove, or modify the attributes of the commands that you are creating. If you were to run the add-in wizard and select the option to create a Tools menu command, the appropriate code to generate a menu item is generated. After running the wizard-generated add-in, if you select the Tools menu, you will see the newly created menu item. But suppose you needed to modify this code to create a combo box instead. If you were to make the appropriate modifications to the code, compile, and then from the Windows Start menu, start a new instance of Visual Studio, the item on the Tools menu will still be a menu item. Why is this?

When the code for a wizard is generated, the appropriate tag is placed into the .addin XML file indicating that the add-in wants to generate a command. When Visual Studio runs, it notices this tag, the add-in is loaded, and the UI setup block of code (the portion of code that checks for the *ext_cm_UISetup* flag in the *connectMode* parameter of *OnConnection*) is run. But this portion of code is run only once; Visual Studio remembers if the add-in's UI setup block has been run, and it will not go through these steps again. Any commands and UI for those commands are persisted across instances of Visual Studio. This is a huge performance benefit, in that your add-in is not loaded and executed every time Visual Studio is run. If, after making the code changes to change the UI type, you were to run the new add-in again from within Visual Studio (by pressing F5 or Ctrl+F5, not from the start menu in Windows), the command will be changed. This happens because the wizard-generated code modifies how the wizard is run when launched from within Visual Studio. If you were to open the properties window for your add-in project and inspect the Debug tab of the properties window, you would notice that a command-line argument named /resetaddin is used. This switch directs Visual Studio to find all the commands and command UI owned by the specified add-in and remove them before starting. Visual Studio will then reload the add-in and execute the UI setup block of the add-in again, causing your commands to be regenerated.

The /resetaddin command-line argument requires a value when it is used. The first possible value is the fully qualified name (in the form of the *Namespace.ClassName*) of the add-in to reset. The second possible value is the asterisk character (*). When this value is used, all commands and their UI for all add-ins are removed. This gives you a way of cleaning up all add-ins at once. You can also use the /resetaddin switch if you are trying to delete an add-in from the system (such as from an uninstall program). First, you will need to delete the .addin file and any .dll files placed on disk for your add-in. Your uninstall program can then issue the command line devenv /resetaddin *Namespace.ClassName* /command File.Exit, where *Namespace.ClassName* is the fully qualified name of the class implementing the

add-in. This will force all the commands to be deleted for the add-in, and the /command File.Exit command line switch will force Visual Studio to close when the commands have been deleted. Because the .addin file is no longer present, the commands for that add-in will not be re-created.

Looking Ahead

In the next chapter, we'll focus on using the object model to create and modify solutions and projects that are loaded into Visual Studio. We'll also look at how to work with those solutions, such as changing how a solution and projects within the solution are compiled into a running program.

Chapter 8

Managing Solutions and Projects Programmatically

In this chapter:

Microsoft® Visual Studio® 2005 is rich with tools to help you manage and complete your programming tasks. One of these tools is the project management system. Projects are where files are created, managed, and compiled to create the resulting program. In this chapter, you'll discover how you can manipulate solutions and projects by using the automation object model.

Working with Solutions

In Visual Studio, a solution is the basic unit of project management. A solution is a container for any number of projects that work together to create the whole of a program. Each project within a solution can contain code files that are compiled to create the program, folders to make managing the files easier, and references to other software components that a project might use. You manage a solution through the Solution Explorer tool window, where you can add, remove, and modify projects and the files they contain. When a solution file is opened, a node is created within Solution Explorer that represents the solution, and each project added to this solution appears as a subnode of the top-level node.

Within the Visual Studio object model, a solution is represented by the *EnvDTE.Solution* object, which you can retrieve using the *Solution* property of the *DTE* object, as shown in the following macro:

```
Sub GetSolution()
    Dim solution As EnvDTE.Solution
    solution = DTE.Solution
End Sub
```

Creating, Loading, and Unloading Solutions

To use the *Solution* object and its methods and properties, you don't need to create or open a solution file from disk. You can use the *Solution* object even though the solution node in Solution Explorer might not be visible. Visual Studio always has a solution open, even if it exists only in memory and not on disk. If you open a solution file from disk and the in-memory solution is not dirty (modified but not saved to disk), this in-memory solution is discarded and the solution on disk is loaded. If the in-memory solution has been modified (such as by having a new or existing project added), when you close it, you'll be prompted to save the solution to disk.

To save a solution programmatically, you can use the method *Solution.SaveAs*; you pass it the full path, including the file name and the .sln file extension, to where the solution should be stored on disk. However, using the *Solution.SaveAs* method might not always work and can generate an exception because you must first save a solution file to disk or load it from an existing solution file on disk before you can use the *SaveAs* method. To allow saving of the solution file, you can use the *Create* method. You use this method to specify information such as where the solution file should be saved and the name of the solution. By combining the *Create* and *SaveAs* methods, you can create and save the solution:

```
Sub CreateAndSaveSolution()
    DTE.Solution.Create("C:\", "Solution")
    DTE.Solution.SaveAs("C:\Solution.sln")
End Sub
```

Once you create a solution file and save it to disk, whether through the user interface or the object model, you can use the *Solution.Open* method to open it. Using the file path given in the *CreateAndSaveSolution* macro, we can open our solution as shown here:

```
DTE.Solution.Open("C:\Solution.sln")
```

When you call this method, the currently open solution is discarded, and the specified solution file is opened. When an open solution is closed to make way for the solution file that is being loaded, the user won't be notified that the current solution is being closed, even if the current solution has been modified. This means that you won't be given the option to save any changes. A macro or an add-in can use the *ItemOperations.PromptToSave* property to offer the option of saving a solution. The *ItemOperations* object, which is accessed from the *DTE.ItemOperations* property, contains various file manipulation methods and properties. One property of this object, *PromptToSave*, displays a dialog box that gives you the option to save modified files and returns a value indicating which button was clicked. This property also saves the files for you if the appropriate user interface button is selected. This property won't show the dialog box if no files need to be saved—it will immediately return a value indicating that you clicked the OK button. You

can combine the *PromptToSave* property with the *Open* method to properly save modified files and open a solution:

```
Sub OpenSolution()
    Dim promptResult As vsPromptResult
    'Offer to save any open and modified files:
    promptResult = DTE.ItemOperations.PromptToSave
    'If the user pressed anything but the Cancel button,
    ' then open a solution file from disk:
    If promptResult <> vsPromptResult.vsPromptResultCancelled Then
        DTE.Solution.Open("C:\Solution.sln")
    End If
End Sub
```

You've learned how to create, save, and open a solution—the only piece of the life cycle of a solution you haven't learned is how to close it. The *Solution* object supports the method *Close*, which you can use to close a solution file. This method accepts one optional *Boolean* parameter, which you can use to direct Visual Studio to save the file when you close it. If you pass the value *true* for this parameter, the solution file is saved before you close it; if you set it to *false*, any changes to the file are discarded.

Enumerating Projects

The *Solution* object is a collection of *Project* objects, and because it is a collection, it has an *Item* method that you can use to find a project within the solution. This method supports the numeric indexing method, as the *Item* method of other collection objects do, but it also supports passing a string to find a project. The string form of the *Solution.Item* method is different from that of other *Item* methods, however. Rather than taking the name of a project, *Solution.Item* requires the unique name of a project. A unique name, as its name indicates, uniquely identifies a project among all other projects within a solution. Unique names are used to index the projects collection because Visual Studio might eventually support loading two projects that have the same name but are located in different folders on disk. (Visual Studio requires that all projects within a solution have a name that is different from all other projects.) Because loading two or more projects with the same name might be allowed in a future version of Visual Studio, the name alone isn't enough to differentiate one project from another when you call the *Item* method. You can retrieve the unique name of a project by using the *Project.UniqueName* property. The following macro retrieves this value for all the projects loaded into a solution:

```
Sub EnumProjects()
    Dim project As EnvDTE.Project
    For Each project In DTE.Solution
        MsgBox(project.UniqueName)
    Next
End Sub
```

The *Solution* object isn't the only collection of all projects that are loaded. The *Solution* object has a *Projects* property, which also returns a collection of the available projects and

works in the same way that the *Solution* object does for enumerating and indexing projects. It might seem redundant to have this same functionality in two places, but the Visual Studio object model team, after performing usability studies, found that developers didn't recognize the *Solution* object as a collection. The team, therefore, added this *Projects* collection to help developers find the list of projects more easily. You can rewrite the *EnumProjects* macro, as follows, so it can use the *Projects* collection:

```
Sub EnumProjects2()
    Dim project As EnvDTE.Project
    For Each project In DTE.Solution.Projects
        MsgBox(project.UniqueName)
    Next
End Sub
```

You can find the list of projects by using the *Solution* and *Projects* collections, but at times you'll need to find the projects that you've selected within the Solution Explorer tree view window. The *DTE.ActiveSolutionProjects* property, when called, looks at the items selected within Solution Explorer. If a project node is selected, the *Project* object for that selected project is added to a list of objects that will be returned. If a project item is selected, the project containing that item is also added to the list returned. Finally, any duplicates are removed from the list, and the list is returned. The following macro demonstrates the use of this property:

```
Sub FindSelectedProjects()
    Dim selectedProjects As Object()
    Dim project As EnvDTE.Project
    selectedProjects = DTE.ActiveSolutionProjects
    For Each project In selectedProjects
        MsgBox(project.UniqueName)
    Next
End Sub
```

Adding Projects to a Solution

While you can use the object model to enumerate projects within a solution, there may be times when you need to add a new or existing project to the solution. You add projects to the solution by using the *AddFromTemplate* and *AddFromFile* methods on the *Solution* object. *AddFromFile* takes a path to an existing project on disk and inserts that project into the solution. *AddFromTemplate* will create a new project within the solution based upon a VSTemplate file—the same templates that we showed you how to create in Chapter 4. Calling this method will invoke the template wizard, causing the project to be copied into a destination folder that you specify and causing all replacement tokens within the files for that project to be replaced with the appropriate values. 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 is 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 solution 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*.

You can find the path to a template that is installed for all users with the method *Solution.GetProjectTemplate*. To find the path to the template, you need to supply the template name that you want to find, as well as the programming language of the template that you want to add. The template name is the .zip file that contains all the files necessary to re-create a project. For example, the Microsoft Visual C#® console application template is named ConsoleApplication.zip, and the language name for Visual C# is CSharp, so a macro that finds this template would look like so:

```
Sub ProjectTemplatePath()
    Dim solution2 As EnvDTE80.Solution2
    Dim CSConsoleTemplatePath As String

    solution2 = CType(DTE.Solution, EnvDTE80.Solution2)
    CSConsoleTemplatePath = solution2.GetProjectTemplate(_
           ConsoleApplication.zip", "CSharp")
    MsgBox(CSConsoleTemplatePath)
End Sub
```

You can create a project based upon a template stored in the My Documents folder, but because there is no automated way of calculating the path to this template, you need to calculate it yourself. When a .zip file is placed into the appropriate folder under the My Documents\Visual Studio 2005\ProjectTemplates folder, and Visual Studio notices the new template (because the New Project dialog box has been shown), it will extract that template into a cache folder at C:\Documents and Settings*username*\Application Data\Microsoft\VisualStudio\8.0\ProjectTemplatesCache. We can use the .NET Framework method

Environment.GetFolderPath to find the first portion of this path, but we will need to construct the rest and add to it the language and template name ourselves to find the full path of the .vstemplate file. If you have a Microsoft Visual Basic® project template named MyTemplate.zip in the correct place, you can find the path to the .vstemplate file with code such as this:

```
Sub UserProjectTemplatePath()
    Dim projectTemplatePath As String
    projectTemplatePath = System.Environment.GetFolderPath( _
            System.Environment.SpecialFolder.ApplicationData)
    projectTemplatePath = System.IO.Path.Combine(projectTemplatesPath, _
            "Microsoft\VisualStudio\8.0\ProjectTemplatesCache")
    'projectTemplatesPath contains the path for all templates, now add to
    '  the path using information specific to the template to find:
    projectTemplatePath = System.IO.Path.Combine(projectTemplatesPath, _
            "Visual Basic\MyTemplate.zip\MyTemplate.vstemplate")
End Sub
```

Capturing Solution Events

As you interact with a solution, Visual Studio fires events that allow an add-in or a macro to receive notifications about which actions you perform. These events are fired through the *SolutionEvents* object, which you can access through the *Events.SolutionEvents* property. You can capture solution events in the usual way—by opening the *EnvironmentEvents* module of any macro project, selecting the *SolutionEvents* object in the left drop-down list at the top of the code editor window, and selecting the event name in the right drop-down list of this window.

Here are the signatures and meanings for the events available for a solution:

- ***void Opened()*** This event is fired just after a solution file has been opened.
- ***void Renamed(string OldName)*** This event handler is called just after a solution file has been renamed on disk. The only argument passed to this handler is the full path of the solution file just before it was renamed.
- ***void ProjectAdded(EnvDTE.Project Project)*** This event is fired when a project is inserted into the solution. One argument is passed to this event handler—the *EnvDTE.Project* object for the project that was inserted.
- ***void ProjectRenamed(EnvDTE.Project Project, string OldName)*** This event is fired when a project within the solution has been renamed. The event handler is passed two arguments. The first is of type *EnvDTE.Project* and is the object for the project that has just been renamed. The second parameter is a string that contains the full path of the project file before it was renamed.
- ***void ProjectRemoved(EnvDTE.Project Project)*** This event is fired just before a project is removed from the solution. This event handler receives as an argument the

EnvDTE.Project object for the project that is being removed. Just as when you use the *BeforeClosing* event, you shouldn't modify the project being removed within this event because the project has already been saved to disk (if you specified that the file be saved) before being removed, and any modifications to the project will be discarded.

- ***void QueryCloseSolution(ref bool fCancel)*** This event is fired just before Visual Studio begins to close a solution file. The handler for this event is passed one argument—a reference to a *Boolean* variable. An add-in or a macro can block a solution from being closed by setting this parameter to *true*, or it can allow the solution to be closed by setting the parameter to *false*. You should take care when you choose to stop the solution from being closed—users might be unpleasantly surprised if they try to close the solution but a macro or an add-in disallows it.
- ***void BeforeClosing()*** This event is fired just before the solution file is about to close but after it has been saved (if you specified the option to save). Because this event is fired after the chance to save the file has passed, the event handler shouldn't make any changes to the solution because those changes will be discarded.
- ***void AfterClosing()*** This event is fired just after the solution file has finished closing.

The sample named SolutionEvents, which is among the book's sample files, demonstrates connecting to each of these events. Once you load this sample, as each event is fired, the add-in displays a message box showing a bit of information about the event that was fired. The *QueryCloseSolution* event handler also offers the option of canceling the closing of the solution. The source code for this add-in sample is shown in Listing 8-1.

Listing 8-1 SolutionEvents.cs, the source code for the solution events add-in

```
namespace SolutionEvents
{
    using System;
    using Microsoft.VisualStudio.CommandBars;
    using Extensibility;
    using EnvDTE;
    using EnvDTE80;
    using System.Windows.Forms;

    public class Connect : Object, IDTExtensibility2
    {
        public Connect()
        {
        }

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

```
            //Set the solutionEvents delegate variable using the
            // DTE.Events.SolutionEvents property:
   solutionEvents =
      (EnvDTE.SolutionEvents)applicationObject.
Events.SolutionEvents;

            //Setup all available event handlers by creating a new
            // instance of the appropriate delegates:
            solutionEvents.AfterClosing += new
               _dispSolutionEvents_AfterClosingEventHandler(AfterClosing);
            solutionEvents.BeforeClosing += new
               _dispSolutionEvents_BeforeClosingEventHandler
(BeforeClosing);
            solutionEvents.Opened += new
_dispSolutionEvents_OpenedEventHandler(Opened);
            solutionEvents.ProjectAdded += new
_dispSolutionEvents_ProjectAddedEventHandler(ProjectAdded);
            solutionEvents.ProjectRemoved += new
_dispSolutionEvents_ProjectRemovedEventHandler
(ProjectRemoved);
            solutionEvents.ProjectRenamed += new
_dispSolutionEvents_ProjectRenamedEventHandler
(ProjectRenamed);
            solutionEvents.QueryCloseSolution += new
               _dispSolutionEvents_QueryCloseSolutionEventHandler
(QueryCloseSolution);
            solutionEvents.Renamed += new
               _dispSolutionEvents_RenamedEventHandler(Renamed);
        }

        public void OnDisconnection(ext_DisconnectMode disconnectMode,
 ref Array custom)
        {
            //The Add-in is closing. Disconnect the event handlers:
            solutionEvents.AfterClosing -= new
_dispSolutionEvents_AfterClosingEventHandler
(AfterClosing);
            solutionEvents.BeforeClosing -= new
_dispSolutionEvents_BeforeClosingEventHandler
(BeforeClosing);
            solutionEvents.Opened -= new
_dispSolutionEvents_OpenedEventHandler(Opened);
            solutionEvents.ProjectAdded -= new
 _dispSolutionEvents_ProjectAddedEventHandler(ProjectAdded);
            solutionEvents.ProjectRemoved -= new
               _dispSolutionEvents_ProjectRemovedEventHandler
(ProjectRemoved);
            solutionEvents.ProjectRenamed -= new
_dispSolutionEvents_ProjectRenamedEventHandler
(ProjectRenamed);
            solutionEvents.QueryCloseSolution -= new
_dispSolutionEvents_QueryCloseSolutionEventHandler
(QueryCloseSolution);
            solutionEvents.Renamed -= new
```

```
_dispSolutionEvents_RenamedEventHandler(Renamed);
        }

        public void OnAddInsUpdate(ref Array custom)
        {
        }

        public void OnStartupComplete(ref Array custom)
        {
        }

        public void OnBeginShutdown(ref Array custom)
        {
        }

        //SolutionEvents.AfterClosing delegate handler:
        public void AfterClosing()
        {
            MessageBox.Show("SolutionEvents.AfterClosing", "Solution Events");
        }

        //SolutionEvents.BeforeClosing delegate handler:
        public void BeforeClosing()
        {
            MessageBox.Show("SolutionEvents.BeforeClosing", "Solution Events");
        }

        //SolutionEvents.Opened delegate handler:
        public void Opened()
        {
            MessageBox.Show("SolutionEvents.Opened", "Solution Events");
        }

        //SolutionEvents.ProjectAdded delegate handler.
        //Display the UniqueName of the project that has been added.
        public void ProjectAdded(EnvDTE.Project project)
        {
            MessageBox.Show("SolutionEvents.ProjectAdded\nProject: " +
 project.UniqueName, "Solution Events");
        }

        //SolutionEvents.ProjectRemoved delegate handler.
        //Display the UniqueName of the project that has been added.
        public void ProjectRemoved(EnvDTE.Project project)
        {
            MessageBox.Show("SolutionEvents.ProjectRemoved\nProject: " +
 project.UniqueName, "Solution Events");
        }

        //SolutionEvents.ProjectRemoved delegate handler.
        //Display the UniqueName of the project that has been renamed,
        // and the full path file before it was renamed.
        public void ProjectRenamed(EnvDTE.Project project, string oldName)
```

```
        {
            MessageBox.Show("SolutionEvents.ProjectRenamed\nProject: " +
 project.UniqueName + "\nOld project name: " + oldName,
 "Solution Events");
        }

        //SolutionEvents.QueryCloseSolution delegate handler.
        //Asks if closing the solution should be canceled.
        public void QueryCloseSolution(ref bool cancel)
        {
            if (MessageBox.Show(
"SolutionEvents.QueryCloseSolution\nContinue with close?",
               "Solution Events", MessageBoxButtons.YesNo) ==
               DialogResult.Yes)
                cancel = false;
            else
                cancel = true;
        }

        //SolutionEvents.QueryCloseSolution delegate handler.
        //Displays the full path the solution before and after it was renamed.
        public void Renamed(string oldName)
        {
            MessageBox.Show(
"SolutionEvents.Renamed\nNew solution name: " +
applicationObject.Solution.FullName +
"\nOld solution name: " + oldName, "Solution Events");
        }

        private DTE2 applicationObject;
        private AddIn addInInstance;

        //The delegate handler variable:
        private EnvDTE.SolutionEvents solutionEvents;
    }
}
```

Is It a Bug When My Events Are Being Disconnected?

Over the years, I've often been asked if there is a bug with events because events can be unexpectedly lost and no longer fire, even if code to disconnect an event is never run. This problem is because of a common programming mistake that reveals itself because of how the garbage collector works in the .NET Framework. Look at the following code, which connects to the solution's *Renamed* event:

```
public void ConnectSolutionEvents()
{
    EnvDTE.SolutionEvents solutionEvents;
    solutionEvents = (EnvDTE.SolutionEvents)
    applicationObject.Events.SolutionEvents;
```

```
    solutionEvents.Renamed += new
        _dispSolutionEvents_RenamedEventHandler(Renamed);
}
```

When this method is called to connect to the *Renamed* event, the *solutionEvents* variable is assigned to an instance of the *SolutionEvents* object. But the *solutionEvents* variable is local to the *ConnectSolutionEvents* method and, as a result, when *ConnectSolutionEvents* returns to the caller, *solutionEvents* is marked as available to be garbage collected. Usually the event fires once or twice, but when the garbage collector starts working, it sees that this variable can be removed from memory and removes it, thus disconnecting the event handler. To make your event handler code work correctly, you should move the *solutionEvents* variable declaration outside the method and to the class scope. This will ensure that the event handler isn't collected until the class is unloaded. Also, note that this behavior applies to all event handlers when they're connected using the .NET Framework, not just the *Solution Renamed* event.

Working with Project Items

Solutions manage a number of projects, and each project manages the files that are built into a program. Each project contains files that can be enumerated and programmed.

Enumerating Project Items

Files within a project are arranged hierarchically. A project can contain any number of files and one or more folders, which themselves can contain additional files and folders. To match this project hierarchy, the project object model is also arranged hierarchically, with the *ProjectItems* collection representing the nodes that contain items and the *ProjectItem* object representing each item within this collection. To enumerate this hierarchy, you use the *ProjectItems* and *ProjectItem* objects. The following macro walks the first level of the hierarchy of the *ProjectItems* and *ProjectItem* objects by using the *Project.ProjectItems* property to obtain the top-level *ProjectItems* object:

```
Sub EnumTopLevelProjectItems()
    Dim projItem As EnvDTE.ProjectItem
    Dim projectProjectItems As EnvDTE.ProjectItems
    Dim project As EnvDTE.Project

    'Find the first project in a solution:
    project = DTE.Solution.Projects.Item(1)
    'Retrieve the collection of project items:
    projectProjectItems = project.ProjectItems
    'Walk the list of items in the collection:
    For Each projItem In projectProjectItems
        MsgBox(projItem.Name)
    Next
End Sub
```

Some items within a project, such as a folder, are both an item within the project hierarchy and a container of other files and folders. Because these folders are both items and collections of items, a folder is represented in the project model hierarchy with both a *ProjectItem* object and a *ProjectItems* object. You can determine whether a *ProjectItem* node is also a container of more *ProjectItem* nodes by calling the *ProjectItem.ProjectItems* property, which returns a *ProjectItems* collection if the node can contain subitems. You can enumerate all the *ProjectItem* and *ProjectItems* objects within a project by writing a recursive macro function such as this:

```
Sub EnumProjectItems(ByVal projItems As EnvDTE.ProjectItems)
    Dim projItem As EnvDTE.ProjectItem
    'Find all the ProjectItem objects in the given collection:
    For Each projItem In projItems
        MsgBox(projItem.Name)
        'And walk any items the current item may contain:
        EnumProjectItems(projItem.ProjectItems)
    Next
End Sub

Sub EnumProject()
    Dim project As EnvDTE.Project
    'Find the first project in a solution:
    project = DTE.Solution.Projects.Item(1)
    EnumProjectItems(project.ProjectItems)
End Sub
```

The *EnumProject* macro first finds the *ProjectItems* collection of a given project, and then it calls the *EnumProjectItems* subroutine, which will find all the *ProjectItem* objects that the collection contains. If the *ProjectItem* object is itself a collection, it will recursively call the *EnumProjectItems* subroutine to display the items it contains.

Folders aren't the only items that can contain a collection of *ProjectItem* objects. Files such as Windows Forms and Web Forms are also collections of files. Each of these file types has associated resource files (in the form of .resx files), and they can also have an associated code-behind file. In the default state, Solution Explorer won't give any indication of whether these files are containers for other files, but you can modify it to show the files that these file types contain. Choose Show All Files from the Project menu to show all form files as expandable in the tree view that makes up Solution Explorer. When the *EnumProject* macro (shown earlier) is run, the *ProjectItem.ProjectItems* property returns a collection that contains the *ProjectItem* objects for these subitems. Code such as the *EnumProject* macro will return the same values whether or not the Show All Files menu command has been chosen. This command affects only the Solution Explorer user interface.

You can combine the techniques for enumerating files and files within folders to find a specific item within a project. Suppose you've created a Windows Forms application solution and modified the project to look like that shown in Figure 8-1.

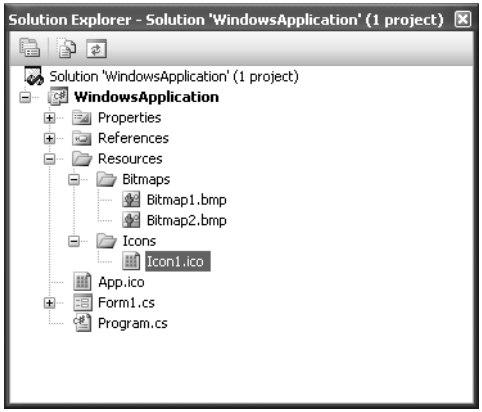


Figure 8-1 A Windows Forms application with nested resources

Using the *ProjectItem* object and *ProjectItems* collection, you can write a macro such as the following to locate the Bitmap1.bmp file:

```
Sub FindBitmap()
    Dim project As EnvDTE.Project
    Dim projectProjectItems As EnvDTE.ProjectItems
    Dim resourcesProjectItem As EnvDTE.ProjectItem
    Dim resourcesProjectItems As EnvDTE.ProjectItems
    Dim bitmapsProjectItem As EnvDTE.ProjectItem
    Dim bitmapsProjectItems As EnvDTE.ProjectItems
    Dim bitmapProjectItem As EnvDTE.ProjectItem

    'Get the project:
    project = DTE.Solution.Item(1)
    'Get the list of items in the project:
    projectProjectItems = project.ProjectItems
    'Get the item for the Resources folder:
    resourcesProjectItem = projectProjectItems.Item("Resources")
    'Get the collection of items in the Resources folder:
    resourcesProjectItems = resourcesProjectItem.ProjectItems
    'Get the item for the Bitmaps folder:
    bitmapsProjectItem = resourcesProjectItems.Item("Bitmaps")
    'Get the collection of items in the Bitmaps folder:
    bitmapsProjectItems = bitmapsProjectItem.ProjectItems
    'Get the item for the Bitmap1.bmp file:
    bitmapProjectItem = bitmapsProjectItems.Item("Bitmap1.bmp")
    MsgBox(bitmapProjectItem.Name)
End Sub
```

You can walk down the tree of the *ProjectItem* and *ProjectItems* hierarchy to find a specific file, but sometimes you might need a quicker and easier way of locating the *ProjectItem* object for a file with a specific file name in a project. You can use the *FindProjectItem* method of the *Solution* object to find an item by passing a portion of the file path to where the file is located on disk. For example, suppose two add-in projects have been created (using the Add-in Wizard) in a folder you created called Addins located in the root of drive C. Each of these

two add-ins, MyAddin1 and MyAddin2, contains a file named Connect.cs. You could use the following macro to locate the Connect.cs file in either project:

```
Sub FindItem()
    Dim projectItem As EnvDTE.ProjectItem
    projectItem = DTE.Solution.FindProjectItem("Connect.cs")
End Sub
```

However, because *FindProjectItem* returns any file that matches this file name, you can't tell which *ProjectItem* will be returned—the *ProjectItem* object for the Connect.cs in MyAddin1 or the *ProjectItem* object for Connect.cs in MyAddin2. To refine the search, you can supply a bit more of the file path as the specified file name, as shown in the following macro, which adds the name of the folder on disk that contains the MyAddin1 version of Connect.cs:

```
Sub FindItemWithFolder()
    Dim projectItem As EnvDTE.ProjectItem
    projectItem = DTE.Solution.FindProjectItem("MyAddin1\Connect.cs")
End Sub
```

Of course, just as you can specify a portion of the path to find the *ProjectItem*, you can use the whole path to zero in on the exact item you want:

```
Sub FindItemWithFullPath()
    Dim projectItem As EnvDTE.ProjectItem
    projectItem = _
        DTE.Solution.FindProjectItem("C:\Addins\MyAddin1\Connect.cs")
End Sub
```

Adding and Removing Project Items

You can add new files to a project in two ways. The first way is to use the *AddFromDirectory*, *AddFromFile*, *AddFromFileCopy*, and *AddFromTemplate* methods of the *ProjectItems* interface (which we'll discuss shortly). The second way is to use the *ItemOperations* object. This object offers a number of file manipulation methods to help make working with files easier. The difference between using the methods of the *ProjectItems* object and the methods of *ItemOperations* is that the *ProjectItems* object gives an add-in or a macro more fine-grained control over where, within a project, the new file is created. The *ItemOperations* object methods are more user-interface oriented; they add the new file to the project or folder that is selected in Solution Explorer, or, if a file is selected, they add the item to the project or the folder containing that file. These features help make macro recording possible. If you start the macro recorder and then add a file into a project, a call to one of the methods of the *ItemOperations* object is recorded into the macro. The selected item is where files are added when the proper method is called.

One method of the *ItemOperations* object, *AddExistingItem*, takes as its only argument the path to a file on disk and adds this file to the selected project or folder within a project. Depending on the type of project, the file might be copied to the project folder before being added, or a reference might be added to the file without copying the file. Visual Basic

and Visual C# projects are folder-based, which means that the project hierarchy shown in Solution Explorer is mirrored on disk, and any files within the project must be in the folder containing the project or in one of its subfolders. Microsoft Visual C++® projects work a little differently: a file that is part of the project can be located anywhere on disk, and it doesn't need to be within the folder containing the project or a child folder of that folder. For instance, suppose a file named file.txt is located in the root directory of the C drive. If we run the macro

```
Sub AddExistingItem()
    DTE.ItemOperations.AddExistingItem("C:\file.txt")
End Sub
```

and the item selected in Solution Explorer is a Visual C# or a Visual Basic project or one of its children, file.txt will be copied into the folder or subfolder containing the project file and then added to the project. But if the selected item is a Visual C++ project, the file will be left in place and a reference will be added to this file.

Whereas *AddExistingItem* inserts a file from disk into a project, *AddNewItem* creates a new file and adds it to the project. This method takes two arguments and has the following method signature:

```
public EnvDTE.ProjectItem AddNewItem(string Item = "General\Text File",
        string Name = "")
```

You can add a new item through the user interface by right-clicking on a project and choosing Add | Add New Item from the shortcut menu, which brings up the Add New Item dialog box.

The Add New Item dialog box is related to the *AddNewItem* method in that the first parameter of *AddNewItem* is the type of file to be added. You can find this file type by using the Add New Item dialog box. Figure 8-2 shows two different Add New Item dialog boxes—the Add New Item dialog box for a Visual C# project (you will also see a similar dialog box for Visual Basic and Microsoft Visual J#® projects) and the dialog box you use when trying to add a new item to a Visual C++ project. If the dialog box you see is of the first type, the name is computed by first taking the language name, such as Visual C#, Visual Basic, or Visual J#, adding the string "*Project Items*\", and appending the name of the file selected within the list view on the right side of the dialog box. This means that the string to add a class file to a Visual C# project would be "Visual C# Project Items\Class". If the dialog that appears looks like that in the second Add New Item dialog box in Figure 8-2, the file type is calculated by taking the path to the item selected in the tree view, with each portion of the path separated by a backslash, and then adding the title of the item in the list on the right side of the dialog box. So, for example, when a Windows Forms file is added to the project, the topmost node of the tree view (Visual C++) is concatenated with the backslash character. Next, the string *UI* is appended to this string (because it is the tree node that contains the Windows Forms item to be added), followed by another backslash. Finally, the name of the item shown in the right panel of the dialog box, the string *Windows Form*, is added,

resulting in the string that can be passed to *AddNewItem*: *Visual C++\UI\Windows Form*. The second argument of this method is simply the name of the file to create, with the file name extension appended. If the file name parameter passed is an empty string, a default file name is generated and used.

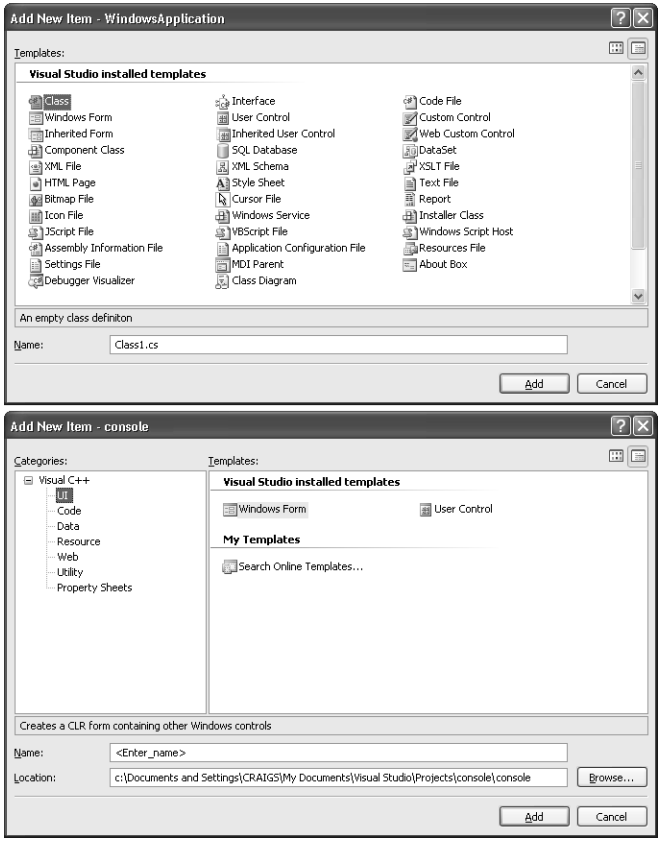


Figure 8-2 The Add New Item dialog box for a Visual C# and a Visual C++ project

The *ItemOperations* object will add new and existing files to the selected project within Solution Explorer, but you will not always want to modify the selection to add an item to a specific solution. The *ProjectItems* collection has a series of methods for adding new and existing files to any project in a solution: *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 file to the project while *AddFromFile* adds a link to the item in whichever folder the file is contained. *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; if the template file has the .vstemplate extension, the template wizard is started and called to add the file to the project.

Here are the signatures, the parameters, and the meanings of the parameters for these methods:

```
EnvDTE.ProjectItem AddFromDirectory(string Directory)
EnvDTE.ProjectItem AddFromFileCopy(string FilePath)
EnvDTE.ProjectItem AddFromFile(string FileName)
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).

Note All of the *Add* methods on both the *Solution* and *ProjectItems* objects will return *null* or *Nothing* (depending upon the programming language you are using) if you pass a .vstemplate file as a template name. This is because a wizard could add 0, 1, or many items to a solution or project. If a wizard does not add any items, *null* or *Nothing* is a logical value to return. If multiple items are added, because the *Add* methods can return only one *Project* or *ProjectItem* object, *null* or *Nothing* is the correct value to return because the wizard would need to select one item arbitrarily to return. If the template wizard were to add one item, it would be confusing if a *Project* or *ProjectItem* object were returned, so, in this case, *null* or *Nothing* is also returned. Also, some of the *Add* methods will allow you to specify a target file name, but the wizard does not necessarily honor that target file name, so finding the item that you suggested a name for may not work. There is another reason for this value to be returned if you pass a .vstemplate as a template file: wizards do not have a mechanism for returning an item to Visual Studio indicating which item has been added, and without the ability to pass back an object, there is nothing to return other than *null* or *Nothing*.

Finding the path to an installed template for adding an item to a project is similar to finding a template for adding a project. For templates installed for all users, rather than using the *GetTemplatePath* method as you do for projects, you use the *GetProjectItemTemplate* for project items. This code uses the *GetProjectItemTemplate* method to find the path to the .vstemplate file for a Visual Basic module:

```
Sub ProjectItemTemplatePath()
    Dim solution2 As EnvDTE80.Solution2
    Dim VBModuleTemplatePath As String
```

```
    solution2 = CType(DTE.Solution, EnvDTE80.Solution2)
    VBModuleTemplatePath = _
           solution2.GetProjectItemTemplate("Module.zip",_
           "VisualBasic")
    MsgBox(VBModuleTemplatePath)
End Sub
```

And just as for project templates installed by the user, you will need to use a brute-force strategy to find the path to the user-installed project item template. Project item templates are extracted into a folder named ItemTemplatesCache when the appropriate Add New Item dialog box is shown. This macro code will file a new project item template, named MyTemplate, for the Visual Basic language:

```
Sub UserProjectItemTemplatePath()
    Dim projectItemTemplatePath As String
    projectItemTemplatePath = System.Environment.GetFolderPath _
           (System.Environment.SpecialFolder.ApplicationData)
    projectItemTemplatePath = System.IO.Path.Combine _
           (projectItemTemplatePath, _
           "Microsoft\VisualStudio\8.0\ItemTemplatesCache")
    'projectItemTemplatePath contains the path for all templates,
    'now add to the path
    '  using information specific to the template to find:
    projectItemTemplatePath = System.IO.Path.Combine _
           (projectItemTemplatePath, _
           "Visual Basic\MyTemplate.zip\MyTemplate.vstemplate")
End Sub
```

You might occasionally need to remove an item that has been added to a project because you no longer need it. The *ProjectItem* object supports two methods for removing items from the project: *Remove* and *Delete*. These two methods both remove an item from the project, but *Delete* is more destructive because it also erases the file from disk by moving it into the computer's Recycle Bin.

Working with Language-Specific Project Objects

The Visual Studio project object model was designed to provide functionality common to all project types. However, some projects can support additional, unique functionality. For example, a Visual C# project has a references node within its project, but a Setup project does not. If you could programmatically add and remove these references, you'd get a lot of flexibility in writing add-ins, macros, and wizards. To enable such project-specific programming, the Visual Studio project object model is extensible, allowing each project type to offer additional methods and properties beyond those defined by the *Project* object.

You can access the specific object type by using the *Object* property of the *Project* object. This property returns an object of type *System.Object*, which you can convert to the object model type supported by a specific language. The most commonly used project extension is the *VSProject* object, which is available for the Visual Basic, Visual J#, or Visual C# project types.

VSProject Projects

VSLangProj.VSProject is the interface that defines extensions to the *EnvDTE.Project* object for Visual J#, Visual Basic, or Visual C# projects. Once you've retrieved the *EnvDTE.Project* interface for one of these project types, you can get to the *VSLangProj.VSProject* interface by calling the *Project.Object* property. The following macro code, which assumes that the first project in the *Projects* collection is a Visual Basic, Visual C#, or Visual J# project, retrieves the *VSProject* object for that project:

```
Sub GetVSProject()
    Dim project As EnvDTE.Project
    Dim vsproject As VSLangProj.VSProject
    project = DTE.Solution.Projects.Item(1)
    vsproject = CType(project.Object, VSLangProj.VSProject)
End Sub
```

References

References are pointers to software components that a project can use to reduce the amount of code a programmer needs to write. A project uses the type of information contained within a reference to display information in the form of IntelliSense® statement completion. A reference also provides information to the compiler for resolving symbols used in programming code. A reference can be an assembly or another project loaded into the solution, and you can create references to COM components by wrapping the COM-object type information library with an interop assembly. You can add references through the user interface by right-clicking on the References node in a Visual Basic or Visual C# project, choosing Add Reference from the shortcut menu, and then selecting a component in the dialog box that appears.

Using the *VSLangProj.References* object, you can enumerate, add, or remove references. To get to the *References* object, use the *VSProject.References* property. For example, the following code retrieves the *References* object and then enumerates the references that have been added to a project:

```
Sub EnumReferences()
    Dim proj As EnvDTE.Project
    Dim vsproj As VSLangProj.VSProject
    Dim references As VSLangProj.References
    Dim reference As VSLangProj.Reference
    proj = DTE.Solution.Projects.Item(1)
    vsproj = proj.Object
    references = vsproj.References
    For Each reference In references
        MsgBox(reference.Name)
    Next
End Sub
```

You add a reference to an assembly by calling the *References.Add* method and passing the path to the assembly. The *Add* method copies the assembly into the project output folder

unless a copy of the assembly with the same version and public key information is stored in the global assembly cache (GAC). This is done so that, when the project output is run or loaded by another assembly, the correct referenced assembly can be loaded. The following macro code adds a reference to an assembly:

```
Sub AddReferenceToAssembly()
    Dim vsproj As VSLangProj.VSProject
    Dim proj As EnvDTE.Project
    proj = DTE.Solution.Projects.Item(1)
    vsproj = CType(proj.Object, VSLangProj.VSProject)
    vsproj.References.Add("C:\Program Files\Common Files\" & _
"Microsoft Shared\MSEnv\PublicAssemblies\extensibility.dll")
End Sub
```

This code finds the *VSProject* object for a project and then adds a reference to the Extensibility.dll metadata assembly—the same assembly that contains the definition of the *IDTExtensibility2* interface, which is used for building add-ins. You can't add assemblies located within the GAC as references to a project because the Visual Basic and Visual C# project systems maintain a separation between the files that are referenced for building against and files that are used during a component's run time.

During development, a component that is compiled by one project in a solution might be needed by a component in another project. You can create a reference from one project to another project by using the *References.AddProject* method. This method accepts a *Project* object and adds a reference to that project, as shown here:

```
Sub AddProjectReference()
    Dim vsproj As VSLangProj.VSProject
    Dim proj As EnvDTE.Project
    'Find the project the reference will be added to:
    proj = DTE.Solution.Projects.Item(1)
    vsproj = CType(proj.Object, VSLangProj.VSProject)
    'Find the referenced project:
    proj = DTE.Solution.Projects.Item(2)
    'Make the project to project reference:
    vsproj.References.AddProject(proj)
End Sub
```

Adding a reference to a COM object requires values that are COM-centric and might not be very intuitive to the non-COM programmer: the type library GUID, or library identifier (LIBID), of the type library that defines the COM component, and the version major and minor values of that type library. Using these values, you can add a reference to the type library of a COM component. When a reference to a COM library is made, the project system will first check to see if a primary interop assembly (PIA) exists for that COM object. If a PIA is found, the PIA will be added to the list of references. If a PIA is not found, the project will automatically create an interop assembly (IA) for that type library and then add a reference to that IA. The only difference between an IA and a PIA is a value in the system registry, PrimaryInteropAssemblyName, which is located underneath the LIBID and version

number for a COM object's type library and points to the .dll file containing metadata describing the types the type library defines. Creating a PIA allows for one assembly file to define all types for one COM object, whereas there can be multiple IAs that handle this interop. If you plan on passing COM interfaces from one .NET program to another through a mechanism such as remoting, you should define a PIA because even though you could generate multiple IAs by wrapping the same type library, the types in one IA are considered distinct from the types in another IA, and an error will be generated. The following macro code adds a reference to the type library for Microsoft Windows Media® Player:

```
Sub AddCOMReference()
    Dim vsproj As VSLangProj.VSProject
    Dim proj As EnvDTE.Project
    proj = DTE.Solution.Projects.Item(1)
    vsproj = CType(proj.Object, VSLangProj.VSProject)
    vsproj.References.AddActiveX( _
        "{22D6F304-B0F6-11D0-94AB-0080C74C7E95}", 1, 0)
End Sub
```

Web References

The .NET Framework not only makes traditional software development easier, but it also makes new software development methodologies possible. One of these new methodologies involves XML Web services. XML Web services enable software development across the Internet by placing software code on a server, which can then be accessed by software that is run on the user's computer. Visual Studio makes connecting desktop software to XML Web services as easy as adding a Web reference. When a reference to an XML Web service is made, a special file written using the Web Services Description Language (WSDL) XML schema is downloaded from the server computer and a proxy class (a class that contains the logic to translate a method or property call from the client computer across the Internet to the server computer) is generated from the WSDL file. This proxy class can then be used to call to the XML Web service.

The following macro adds a Web reference to a project. It retrieves the *VSProject* object for a project and then calls the *AddWebReference* method with the URL for the XML Web service. This example uses the TerraServer Web service provided by Microsoft, which offers detailed geographic information and satellite images for the United States. This Web service is located at *http://terraserver.microsoft.com/TerraService2.asmx*

```
Sub AddTerraServerWebRef()
    Dim vsProj As VSLangProj.VSProject
    Dim serviceURL As String
    'Set the URL to the TerraServer web service
    serviceURL = "http://terraserver.microsoft.com/TerraService2.asmx"
    'Find the VSProject for a project
    vsProj = DTE.Solution.Projects.Item(1).Object
    'Add the web reference
    vsProj.AddWebReference(serviceURL)
End Sub
```

When this Web reference is made, the WSDL file describing the XML Web service is downloaded from the server computer, and the proxy class for the service is generated and automatically added to the project. This class is placed in a namespace defined by the Web protocol and server name, but in reverse order. So, for example, if the XML Web service were located at *www.microsoft.com*, the namespace for the service would be *com.microsoft.www*. In this example, TerraServer is located at the server URL *terraserver.microsoft.com* so the namespace used is *com.microsoft.terraserver*. Once a reference to an XML Web service has been added to a project, using that service is as easy as calling methods on the generated proxy class.

Imports

To make the programmer's life easier, Visual Basic and Visual C# source code can contain *using* and *Imports* statements to shorten the identifiers used to access the namespace defined by a library of code. For example, to display a message box, you could use the longer, more specific identifier to resolve to a class name:

```
System.Windows.Forms.MessageBox.Show("Hello World")
```

But if this code were repeated a number of times, you'd have to type the namespace identifier over and over, which could lead to programming errors. You can use an *Imports* statement in Visual Basic to shorten what you have to type:

```
Imports System.Windows.Forms
```

Later in the program, you can use this shorter form of the code:

```
MessageBox.Show("Hello World")
```

Visual Basic also allows you to enter *Imports* statements through a project's Properties window rather than by typing the *Imports* statement into the source code. By using the project properties dialog box instead of typing the statement into code, you can make the import available for all the files within the project, not just the file that uses the *Imports* statement. Using the *VSProject.Imports* collection, you can enumerate, remove, and add imports for the entire project. The following macro adds the *System.XML* namespace to a Visual Basic project:

```
Sub AddSystemXMLImport()
    Dim vsProj As VSLangProj.VSProject
    vsProj = DTE.Solution.Projects.Item(1).Object
    vsProj.Imports.Add("System.Xml")
End Sub
```

Note The *Imports* object is valid only for the Visual Basic project type. Any attempt to access this object for a Visual C# or Visual J# project will return a *null* or *Nothing* value.

ProjectProperties

Each project has a number of options associated with it that allow you to control how you interact with that project. You can set these options under the Common Properties node of a project's Property Pages dialog box. The options include the name of the component that the compiler should build, the kind of project to be generated (an .exe or a .dll), and layout options for the HTML designer. You can also set these options programmatically by using the *Properties* property of the *Project* object. This property returns the same *Properties* object that's used throughout Visual Studio to set options. The following macro walks the list of properties available to a project as well as the values and types of each property:

```
Sub WalkVSProjectProperties()
    Dim project As EnvDTE.Project
    Dim properties As EnvDTE.Properties
    Dim [property] As EnvDTE.Property
    Dim owp As InsideVSNET.Utilities.OutputWindowPaneEx
    owp = New InsideVSNET.Utilities.OutputWindowPaneEx(DTE, _
        "Project properties")
    project = DTE.Solution.Projects.Item(1)
    properties = project.Properties
    For Each [property] In properties
        owp.WriteLine("Name: " + [property].Name)
        owp.WriteLine("Value: " + [property].Value.ToString())
        owp.WriteLine("Type: " + [property].Value.GetType().FullName)
        owp.WriteLine()
    Next
End Sub
```

You can use this *Property* object to read the values of properties and to set the properties for a project. The following macro demonstrates this. It sets the icon to use for a project when it is compiled. This code assumes that an icon named Icon.ico is located in the folder containing the project file.

```
Sub SetProjectIcon()
    Dim project As EnvDTE.Project
    Dim [property] As EnvDTE.Property
    Dim projectPath As String
    project = DTE.Solution.Projects.Item(1)
    'Get the Property object for the icon:
    [property] = project.Properties.Item("ApplicationIcon")
    'Construct the path to the icon based off of the
    ' project path:
    projectPath = project.FullName
    projectPath = System.IO.Path.GetDirectoryName(projectPath)
    projectPath = projectPath + "\Icon.ico"
    'Set the icon for the project:
    [property].Value = projectPath
End Sub
```

Using Visual Studio Utility Project Types

To help you more easily maintain files within a solution, Visual Studio makes available various utility projects. These utility projects allow you to keep track of files that are not part of any other project that is loaded into a solution. Because any file type can be stored within these projects, such as program source files or Microsoft Word documents, these projects can't be compiled into a program. And because utility projects are not associated with any particular programming language, they are available to all users of Visual Studio and don't require Visual Basic, Visual C#, or Visual C++ to be installed.

Miscellaneous Files Project

When you're working with a solution, you might need to open files that are not part of an existing project. When you open such a file, it is automatically added to a project called Miscellaneous Files. A project file isn't created on disk for this project, as with other project types, but you get a convenient way of locating files that are open but are not part of any other project that is open within the solution. You can think of the Miscellaneous Files project as a list of most recently used open documents—when you open a file, an item for that file is added to the project, and when you close the file, it is removed. By default, the Miscellaneous Files project and the files it contains don't appear in the Solution Explorer tree hierarchy, but you can easily make them visible by opening the Tools Options dialog box, selecting the Environment | Documents node, and selecting the Show Miscellaneous Files In Solution Explorer check box.

The Miscellaneous Files project has a unique name associated with it that, unlike with other projects, doesn't change over time. This name, <MiscFiles>, is defined by the constant *vsMiscFilesProjectUniqueName*. The following macro retrieves the *Project* object for the Miscellaneous Files project:

```
Sub FindMiscFilesProject()
    Dim project As EnvDTE.Project
    Dim projects As EnvDTE.Projects
    projects = DTE.Solution.Projects
    project = projects.Item(EnvDTE.Constants.vsMiscFilesProjectUniqueName)
End Sub
```

When the first file is opened within the Miscellaneous Files project, an item is added to the *Solution.Projects* collection that implements the *Project* interface. It works just as the *Project* interface implemented by projects such as Visual Basic or Visual C#, except that a few of the properties will return *null* or *Nothing* or throw a *System.NotImplementedException* when called. Table 8-1 lists the methods and properties of the *Project* object that return a meaningful value for the Miscellaneous Files project and the *ProjectItem* and *ProjectItems* objects contained within this project.

Table 8-1 Methods and Properties of the *Project*, *ProjectItems*, and *ProjectItem* Objects

Project	ProjectItems	ProjectItem
DTE	*DTE*	*DTE*
ProjectItems	*Parent*	*Collection*
Name (read-only)	*Item*	*Name* (read-only)
UniqueName	*GetEnumerator / _NewEnum*	*FileCount*
Kind	*Kind*	*Kind*
FullName	*Count*	*FileNames*
	ContainingProject	*SaveAs*
		Save
		IsOpen
		Open
		Delete
		Remove
		ExpandView
		ContainingProject
		IsDirty

You can add new files to the Miscellaneous Files project by using the *ItemOperations.NewFile* method, which has the following method signature:

```
public EnvDTE.Window NewFile(string Item = "General\Text File",
    string Name = "", string ViewKind =
    "{00000000-0000-0000-0000-000000000000}")
```

By applying the techniques we used earlier to calculate the first parameter for the *ItemOperations.AddNewItem* method, we can find the value that should be passed to the *NewFile* method. The second parameter also has the same meaning as the second parameter of the *ItemOperations.AddNewItem* method—the name of the file (with extension) that is to be added—and if the empty string is passed, a default name is calculated. The last argument specifies which view the file should be opened in when it is added. These values can be found within the *EnvDTE.Constants* class and begin with the name *vsViewKind*.

Solution Folders

The Miscellaneous Files project lists files that are temporarily open in an editor, and when those files are closed, they are removed from that project. But what if you want to associate a file with a solution, not have that file built as part of a project, and have that file stay within

your solution when the editor window for that file has been closed? Solution folders provide a way for you to satisfy these requirements and more. Solution folders can be created by right-clicking on the solution node within the Solution Explorer tool window, and choosing Add | New Solution Folder. Solution folders can also be created within existing solution folders by right-clicking on a solution folder and selecting Add | New Solution Folder. Because solution folders can be nested within one another, you can create a hierarchy of folders, each containing files on disk. Often I need to add a bunch of files to a solution that are not part of a project, and, to keep those files organized, I like to mirror the folder structure on disk with solution folders. Not only can you organize files on disk with solution folders, but you can also use them to organize projects. If your solution contains many projects, you can create a solution folder that contains, for example, all the Web applications in your project, one solution folder for all class libraries, etc. New or existing projects can be added to a solution folder from the context menu for a solution folder, or if the project you want to move is located under the solution node in the Solution Explorer tool window, you can just drag the project into the appropriate solution folder.

A Solution Folder also has a programmatic interface to this functionality. To create a solution folder within a solution, simply call the *Solution2.AddSolutionFolder* method, supplying a name for the new folder:

```
Sub CreateSolutionFolder()
    Dim solution2 As EnvDTE80.Solution2
    solution2 = CType(DTE.Solution, EnvDTE80.Solution2)
    solution2.AddSolutionFolder("My Folder")
End Sub
```

The *AddSolutionFolder* method returns an *EnvDTE.Project* object, which works as any other *Project* object, such as that available from Visual C# or Visual Basic projects, except you will find that many methods, such as the *Save* method, will not work for a solution folder because it has no meaning for that project type. The *Object* property on the *Project* interface for a solution folder returns an object that you can then cast into the interface *EnvDTE80.SolutionFolder*. The *SolutionFolder* interface also has a method named *AddSolutionFolder*, which will allow you to create a nested folder. This macro is a modified version of the one to add a folder to the solution, but it will also create a nested folder:

```
Sub CreateSolutionFolder()
    Dim solution2 As EnvDTE80.Solution2
    Dim project As EnvDTE.Project
    Dim solutionFolder As EnvDTE80.SolutionFolder
    solution2 = CType(DTE.Solution, EnvDTE80.Solution2)
    project = solution2.AddSolutionFolder("MyFolder")
    solutionFolder = CType(project.Object, EnvDTE80.SolutionFolder)
    solutionFolder.AddSolutionFolder("My Other Folder")
End Sub
```

The *SolutionFolder* interface will also allow you to programmatically add projects or files. The method *AddFromFile* will take the path to an existing project file (a file with an extension such as .csproj, .vbproj, or .vcproj), and add it to a solution folder. Here is an example of the

use of this method, which will first create a solution folder named MyFolder and then will add an existing Visual C# project named ConsoleApplication to that solution folder:

```
Sub ProjectAdd()
    Dim project As EnvDTE.Project
    Dim solutionFolder As EnvDTE80.SolutionFolder
    Dim solution2 As EnvDTE80.Solution2

    solution2 = CType(DTE.Solution, EnvDTE80.Solution2)
    project = solution2.AddSolutionFolder("MyFolder")
    solutionFolder = project.Object
    solutionFolder.AddFromFile("C:\Project\ConsoleApplication1.csproj")
End Sub
```

The *AddFromTemplate* method takes a path to a .vstemplate file, a .vsz file, or an existing project to clone; a path to store the project in; and a name of the new project to create. *AddFromTemplate* then generates a new project based upon the wizard or the existing project. This macro uses the *GetProjectTemplate* method of the *Solution2* interface to find the path to the Visual C# Console Application template, and then creates a project based upon this template within a new solution folder.

```
Sub NewProjectAdd()
    Dim project As EnvDTE.Project
    Dim solutionFolder As EnvDTE80.SolutionFolder
    Dim solution2 As EnvDTE80.Solution2
    Dim CSConsoleTemplatePath As String

    solution2 = CType(DTE.Solution, EnvDTE80.Solution2)
    project = solution2.AddSolutionFolder("MyFolder")
    solutionFolder = project.Object

    CSConsoleTemplatePath = solution2.GetProjectTemplate _
           ("ConsoleApplication.zip", "CSharp")
    solutionFolder.AddFromTemplate(CSConsoleTemplatePath, _
           "C:\Projects\TestProject", "NewProject")
End Sub
```

Adding an existing or new file to a solution folder is done within the *Project* object by using the *AddFrom* methods, just as you would add an item to a Visual C# or Visual J# project.

Once you have a solution folder with items located within that folder, you can use the folder as a *Project* object to locate an item contained within it. For example, suppose you have a solution folder with a project loaded into it. The following macro finds the *Project* for the solution folder and then obtains the collection of *ProjectItems* for the solution folder. Even though a project within a solution folder is a project, it is still an item, and so the solution folder allows you to get to a list of *ProjectItem* objects by using the *ProjectItems* property. Once you have this *ProjectItem* object, you can call the *Object* property to retrieve the object specific to that project item node, which is a project.

```
Sub FindProjectInSolutionFolder()
    Dim slnFolderProject As Project
```

```
    Dim containedProject As Project
    Dim projectAsAProjectItem As ProjectItem

    'Find the solution folder. This code expects one folder and
    '  no other items within the solution
    slnFolderProject = DTE.Solution.Projects.Item(1)

    'Find the project item for the project in the folder
    projectAsAProjectItem = slnFolderProject.ProjectItems.Item(1)

    'Convert the project item into a Project
    containedProject = projectAsAProjectItem.Object
    MsgBox(containedProject.Name)
End Sub
```

This is the only way that you can get to a *Project* object for a project that is located within a solution folder. The *Solution.Projects.Item* method will not return a project's object for a project in a solution folder because the *Projects.Item* method looks only at the projects directly underneath the solution node in Solution Explorer.

If you have a reference to a *Project* object, either a project such as a Visual J# project or a *Project* object for a solution folder, and that project is contained within a solution folder, you can find the *ProjectItem* object of that project within the solution folder by using the *ParentProjectItem* property. This macro will find a *Project* object within a solution folder and then walk back up the hierarchy to find the solution folder that contains it.

```
Sub FindParentProject()
    'First, find the project nested in the solution folder:
    Dim nestedProject As Project
    Dim solutionFolder As Project

    solutionFolder = DTE.Solution.Projects.Item(1)
    nestedProject = solutionFolder.ProjectItems.Item(1).Object

    'Now, find the solution folder parent of the nested project
    Dim parentProjectItem As ProjectItem
    Dim parentProject As Project
    parentProjectItem = nestedProject.ParentProjectItem
    parentProject = parentProjectItem.ContainingProject

    'Make sure the parent project and the solution folder are the same
    MsgBox(parentProject.UniqueName = solutionFolder.UniqueName)
End Sub
```

Unmodeled Projects

All the project types we've discussed so far have implemented a *Project* object that can be used by a macro or an add-in. However, a few project types, such as a database project or a project that has been unloaded by using the Project | Unload Project command, don't implement the *Project* object themselves. To allow some programmability for these project types, Visual Studio supports the *unmodeled* project type. An unmodeled project provides

an implementation of the *Project* object that supports only the properties common among all project types, which are *DTE*, *Kind*, and *Name*. All other properties and methods on this implementation of the *Project* object return values that have no useful meaning or generate an exception when called and shouldn't be used by a macro or an add-in. You can distinguish an unmodeled project from other project types by checking the *Project.Kind* property, which returns the constant *EnvDTE.Constants.vsProjectKindUnmodeled* if the project is an unmodeled project. The following macro enumerates all the projects loaded into a solution and determines which ones are unmodeled:

```
Sub FindUnmodeledProjects()
        Dim project As EnvDTE.Project
        For Each project In DTE.Solution.Projects
            If (project.Kind = EnvDTE.Constants.vsProjectKindUnmodeled) Then
                MsgBox(project.Name + " is unmodeled")
            End If
        Next
End Sub
```

Project and Project Item Events

Just as a solution fires events to allow an add-in or macro to respond to the actions the user is performing, the various project types also fire events so that an add-in or a macro can be informed of what the user is doing. You connect to the events fired by the different project types in different ways, but each project type supports the same interfaces used to handle the event invocations. Each project fires two classes of events: actions performed with the project and actions performed with the items within those projects. Here are the events and the signatures that are called when a project is added, removed, or renamed within a solution:

```
void ItemAdded(ByVal Project As EnvDTE.Project)
void ItemRemoved(ByVal Project As EnvDTE.Project)
void ItemRenamed(ByVal Project As EnvDTE.Project,
    ByVal OldName As String)
```

These are the signatures of events fired when a project item is added to, removed from, or renamed within a project:

```
void ItemAdded(ByVal ProjectItem As EnvDTE.ProjectItem)
void ItemRemoved(ByVal ProjectItem As EnvDTE.ProjectItem)
void ItemRenamed(ByVal ProjectItem As EnvDTE.ProjectItem, _
    ByVal OldName As String)
```

To connect to these events, you can use the *Events2* object's *ProjectItemsEvents* and *ProjectsEvents* properties. These properties will return an instance of an *EnvDTE.ProjectItemsEvents* or *EnvDTE.ProjectsEvents* object, which you can then use to receive event notifications from all projects or project items open within a solution. As new projects are added to the solution, or as new items are added to a project, the event handlers for your implementation of these interfaces are called. To connect to one of these events from the

Macros IDE, simply open the EnvironmentEvents module of a macro project, and in the left drop-down at the top of the EnvironmentEvents source code window, select either the ProjectsEvents or ProjectItemsEvents item. Then in the right drop-down, select the event that you want to connect to.

You can connect to these events for all projects open in a solution and you can connect to them on a project type–specific basis. This means that if you want to connect to events on just Visual Basic projects loaded into a solution, or just on Visual C# projects, you can. Rather than using the *Events2.ProjectItemsEvents* and *Events2.ProjectEvents* methods to connect to these language type–specific events, you can use the *Events.GetObject* method, passing in a special string indicating the project type you want to connect to. But first, you need to set up the event variables to handle the event. You declare the event variable within a macro project by adding the following code to the *EnvironmentEvents* module. (This example connects to the Visual C# project events.)

```
<System.ContextStaticAttribute()> _
Public WithEvents csharpProjectItemsEvents As EnvDTE.ProjectItemsEvents
```

When you enter this code, an entry appears in the left drop-down list at the top of the code for the *EnvironmentEvents* macro module. Select the entry to fill the right drop-down list with the events for this object, and select each event to create the code necessary for capturing that event. At this point, the event handler won't be invoked for Visual C# projects because the event variable, *csharpProjectItemsEvents*, has yet to be set to an instance of a *ProjectItemsEvents* object. To set this variable to an instance of the correct event object, create a handler for *DTEEvents.OnStartupComplete* and place within it the code to connect to the event, much as you would within an add-in:

```
Private Sub DTEEvents_OnStartupComplete() _
    Handles DTEEvents.OnStartupComplete

    csharpProjectItemsEvents = _
        DTE.Events.GetObject("CSharpProjectItemsEvents")
End Sub
```

With this event handler in place, when Visual Studio is closed and then restarted, the *OnStartupComplete* handler will be invoked, which will cause the event variable to be connected. Of course, you can insert this same code into a macro and run the macro; this way you will not need to restart Visual Studio for the event variable to be set. Here's an example of such a macro:

```
Sub ConnectCSharpProjectItemsEvents()
    csharpProjectItemsEvents = _
        DTE.Events.GetObject("CSharpProjectItemsEvents")
End Sub
```

You can connect to the project and project item events for project types other than the Miscellaneous File and Solution Items projects by changing the string passed to the *Events.GetObject* method. For example, to connect to Visual Basic project and project

item events, you can use the strings *VBProjectsEvents* and *VBProjectItemsEvents*. You can use the strings *VJSharpProjectsEvents* and *VJSharpProjectItemsEvents* to connect to events thrown by a Microsoft Visual J# project, and you can use *eCSharpProjectsEvents* and *eCSharpProjectItemsEvents* to capture events thrown by a Visual C# smart device application. You can use *eVBProjectsEvents* and *eVBProjectItemsEvents* to capture events thrown by a Visual Basic smart device application. The ProjectEvents sample demonstrates how to connect to all these project and project item events. It connects to the events provided by each project type, and as each event is fired, a message box is displayed containing information about that event.

Managing Build Configurations

Editing and manipulating a project is an important part of the development process, but most of your time is spent building and compiling, not moving around files within a project. Visual Studio provides an object model for building a solution and controlling how the projects contained within that solution should be compiled. The root object for controlling how a solution should be built is named *SolutionBuild*; you access it by calling the *Solution.SolutionBuild* property, and you control how each project within the solution should be built by using the *ConfigurationManager* object, which is accessed through the *Project.ConfigurationManager* property.

Manipulating Solution Settings

Visual Studio uses solution configurations to manage how a solution is built. A *solution configuration* is a grouping of project configurations that describe how the projects within the solution should be built. A *project configuration*, in the simplest terms, tells the various compilers how to create the code for a project. Each project can contain multiple project configurations that you can switch between within the solution configuration to control how the compilers build the code. The most common solution and project configurations are debug and release, which cause a project to be built with debugging information and with code optimizations, respectively. When a Windows Forms project is first created, Visual Studio creates the debug solution configuration containing the Windows Forms debug project configuration and the release solution configuration containing the release Windows Forms project configuration. You can create new solution configurations that contain any of the available project configurations or new project configurations that can be loaded into any solution configuration.

SolutionConfiguration and *SolutionContext* Objects

Solution configurations are represented in the object model through the *SolutionConfigurations* collection, which contains *SolutionConfiguration* objects. Because the *SolutionConfigurations* object is a collection, you can use the standard techniques for enumerating this collection and use the *Item* method to find a specific *SolutionConfiguration* object by name. To create new solution configuration, you use the *SolutionConfigurations.Add*

method, which makes a copy of an existing solution configuration and then renames it to the specified name. The signature of this method is

```
public EnvDTE.SolutionConfiguration Add(string NewName,
    string ExistingName, bool Propagate)
```

Here are the arguments that are passed to this method:

- ***NewName*** This is the name of the new solution configuration. It can't be the same as any existing solution configuration name, and it must follow the file system's file-naming rules. (It can't contain characters such as \, /, :, *, ?, ", <, or >.)
- ***ExistingName*** This is either the name of an existing solution configuration that is copied to create the new solution configuration or the string "*<Default>*". If the name "*<Default>*" is used, the currently active solution configuration is used as the source of what is copied.
- ***Propagate*** If this parameter is *true*, when the new solution configuration is created, a copy of each project configuration referenced by the solution configuration is made and assigned the same name as the new solution configuration, and each of these copies of project configurations is loaded into the new solution configuration. If this parameter is *false*, the new solution configuration is created and the same project configurations that were assigned to the solution configuration source are assigned to the new solution configuration.

The *SolutionConfiguration* object has one method, *Activate*, and one property of note, *SolutionContexts*. When a build is performed, whether through the user interface or through the object model by using the *SolutionBuild.Build* method, the currently active *SolutionConfiguration* is the configuration that is built. Therefore, activating a particular solution configuration by using the *Activate* method causes any build actions to build the active solution configuration. The other item of importance is the *SolutionContexts* property. As discussed earlier, a *SolutionConfiguration* is a container of the projects within a solution and the project configuration associated with that solution configuration. The *SolutionConfiguration.SolutionContexts* property returns a *SolutionContexts* collection, containing those projects and the configuration of each project to build.

To set the project configuration that is built when the solution is built, you can change the *SolutionContext* object's *ConfigurationName* to any project configuration name that the project supports. The following macro changes the debug solution configuration to build the release version of a project that is loaded into the solution:

```
Sub ChangeProjectConfiguration()
    Dim solutionBuild As EnvDTE.SolutionBuild
    Dim solutionCfgs As EnvDTE.SolutionConfigurations
    Dim solutionCfg As EnvDTE.SolutionConfiguration
    Dim solutionContext As EnvDTE.SolutionContext
    'Find the debug solution configuration:
    solutionBuild = DTE.Solution.SolutionBuild
    solutionCfgs = solutionBuild.SolutionConfigurations
```

```
    solutionCfg = solutionCfgs.Item("Debug")
    'Retrieve the solution context for the first project:
    solutionContext = solutionCfg.SolutionContexts.Item(1)
    'Change the debug solution context to build the
    ' Release project configuration:
    solutionContext.ConfigurationName = "Release"
    'Reset the build flag for this context:
    solutionContext.ShouldBuild = True
End Sub
```

You can modify a *SolutionContext* to set the project configuration that should be built for a particular solution configuration and you can also set values such as that specifying whether the project configuration should be built. This is done in the next-to-last line of the preceding macro, where the *ShouldBuild* property is set to *true*. In this macro, this property must be set because, as is expected, when the debug solution configuration is first created, it doesn't contain the release project configuration. It, therefore, isn't set to build for that solution configuration, so when the debug solution configuration is set to build the release project configuration, that "do not build" state is carried along with it.

StartupProjects

When you start a solution running (usually by pressing the F5 key), the project builder first verifies that all the projects that need to be built are up-to-date, and then it starts walking the list of projects that are set as startup projects, running each project in turn. You can set the list of startup projects through the user interface by right-clicking on the solution node in Solution Explorer and then choosing Set StartUp Projects from the shortcut menu. You'll see the Solution Property Pages dialog box (shown in Figure 8-3), in which you can set the startup projects for a solution containing four Windows Forms applications.

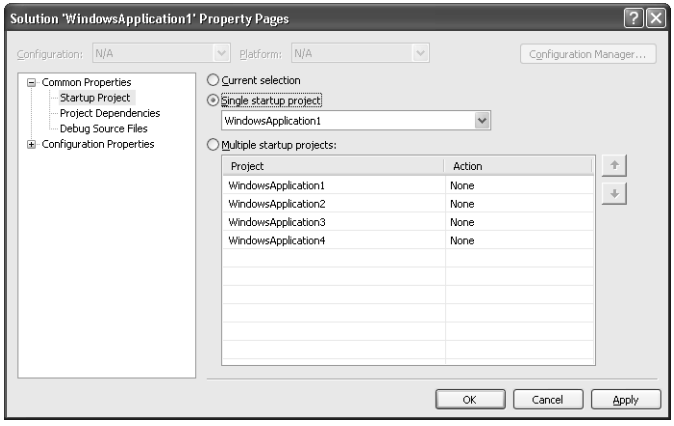


Figure 8-3 Setting the projects that will start when you run a solution

You can also set startup projects through the object model by using the *SolutionBuild.StartupProjects* property. This property is set to a value of type *System.Object*, which is packed with the projects to start when you run a solution. The value passed to the *StartupProjects*

property can take two forms: a single string that is the unique name of a project (which will set one single project to run) or an array of *System.Object* (which will be filled with one or more project unique names and will cause multiple projects to be run).

For example, suppose an open solution contains two projects, each of them to be designated as a startup project. You can use code such as the following to set these projects as startup projects:

```
Sub SetStartupProjects()
    Dim startupProjects(1) As Object
    startupProjects(0) = DTE.Solution.Projects.Item(1).UniqueName
    startupProjects(1) = DTE.Solution.Projects.Item(2).UniqueName
    DTE.Solution.SolutionBuild.StartupProjects = startupProjects
End Sub
```

If only one project should be set as a startup project, the code looks like this:

```
Sub SetStartupProject()
    Dim startupProject As String
    startupProject = DTE.Solution.Projects.Item(1).UniqueName
    DTE.Solution.SolutionBuild.StartupProjects = startupProject
End Sub
```

When you set the startup projects, you must be careful to supply only buildable projects. If one of the projects supplied to *SolutionBuild.StartupProjects* is, for example, the unique name for the Miscellaneous Files project or the Solution Items project, an error is generated.

Project Dependencies

When you work with a solution that contains multiple projects, the components built by one project might rely on the output of another project. An example of this is a control project called UserControl, which is placed on the form of a Windows Forms application called WinForm. Because changes to the UserControl project might affect how that control is used by the Windows Forms project, the UserControl project must be compiled before the WinForm project is compiled. To enforce this relationship between the two projects, you can create a project dependency. The dependencies between two or more projects can be depicted using a dependency graph; the dependency graph for the projects WinForm and UserControl is shown in Figure 8-4. The arrow is pointing to the project that another project is dependent on.

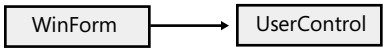


Figure 8-4 A dependency graph showing a WinForm project dependent on a UserControl project

Suppose we add a new project to the solution—a class library called ClassLib that implements functionality used by both the WinForm and the UserControl projects. A dependency graph for this solution is shown in Figure 8-5.

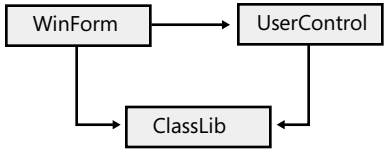


Figure 8-5 The dependency graph for three projects

You can see in this dependency graph that the WinForm project can't be built until the UserControl and ClassLib projects have been built. The UserControl project relies on only the ClassLib project being built first. When a build of this solution is started, if the build system chooses the UserControl project to start building first, the ClassLib project builds. If the build system chooses the ClassLib project first, because it does not have any dependencies, it can build immediately without needing to build any other projects. When the UserControl project is built, the ClassLib project isn't built again because it is up-to-date. Because it relies upon the output of the other two projects, the last project to be built is the WinForm project because it relies on the output of the other two projects.

A problem can occur with a dependency graph if you create a *cyclic dependency*, in which one or more projects are mutually dependent. Suppose the WinForm project relies on the UserControl project, the UserControl project relies on the ClassLib project, and the ClassLib project relies on the WinForm project. The cycle shown in Figure 8-6 is generated.

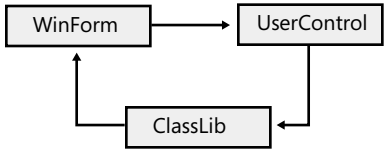


Figure 8-6 A dependency graph of three projects with a cycle

If the WinForm project is built, the build of the UserControl project is triggered because of the dependency. Building the UserControl project causes the building of the ClassLib project, which is dependent on the WinForm project. If the Visual Studio build system were unable to detect this cycle, the loop would continue forever in an attempt to find the first project to build. But Visual Studio is smart enough to detect dependency cycles, and it disallows them.

You can create dependencies between projects through the user interface by choosing Project | Project Dependencies, which will display the Project Dependencies dialog box (shown in Figure 8-7). The dialog box shows all the projects that can be set as a dependency for the UserControl project. The WinForm check box is shaded because a dependency is set from the WinForm project to the UserControl project, and Visual Studio won't allow a cycle between the WinForm project and the UserControl project to be created.

You can also set build dependencies through the object model. The *SolutionBuild.BuildDependencies* property returns a *BuildDependencies* object, which is a collection of *BuildDependency* objects. You can index this collection by using the *Item* method—you can

pass a numeric index, an *EnvDTE.Project* object, or the unique name of a project. Each project in the solution has its own *EnvDTE.BuildDependency* object, whose *RequiredProjects* property you can use to add, remove, or retrieve dependencies for a project. The following macro displays in the Output window the available projects in the open solution, as well as all the projects it depends on.

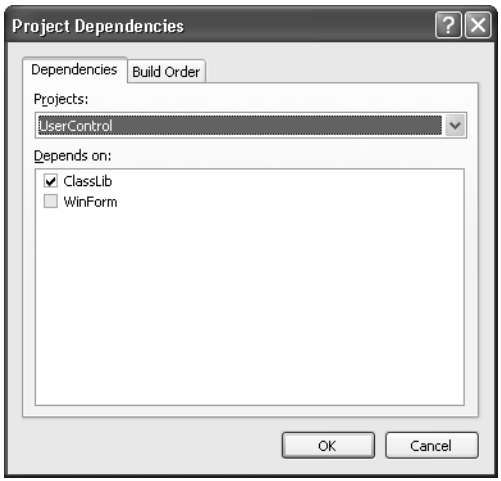


Figure 8-7 Setting project dependencies

```
Sub Depends()
    Dim projectDep As EnvDTE.BuildDependency
    Dim project As EnvDTE.Project
    Dim owp As New InsideVSNET.Utilities.OutputWindowPaneEx(DTE,
        "Build dependencies")

    For Each projectDep In DTE.Solution.SolutionBuild.BuildDependencies
        Dim reqProjects As Object()

        owp.Write("The project ")
        owp.Write(projectDep.Project.Name)
        owp.WriteLine(" relies on:")
        reqProjects = projectDep.RequiredProjects
        If (reqProjects.Length = 0) Then
            owp.WriteLine(vbTab + "<None>")
        Else
            For Each project In reqProjects
                owp.WriteLine(vbTab + project.Name)
            Next
        End If
        owp.WriteLine()
    Next
End Sub
```

Using the *BuildDependency* object, you can create a macro or an add-in that sets up the dependencies between two or more projects. Suppose, using our current example, that

a solution with the projects WinForm, UserControl, and ClassLib is loaded and no dependencies have been set. The *BuildDependency* object supports three methods for modifying the projects that a project is dependent on: *AddProject*, *RemoveProject*, and *RemoveAllProjects*. *AddProject* and *RemoveProject* accept the unique name of a project that should be added or removed as a dependency for a specific project. *RemoveAllProjects* takes no arguments and removes all project dependencies. The following macro, *SetDependencies*, builds the correct dependencies for the three-project solution to conform to the dependency graph shown in Figure 8-5:

```
Sub SetDependencies()
    Dim buildDependencies As EnvDTE.BuildDependencies
    Dim buildDependency As EnvDTE.BuildDependency
    Dim project As EnvDTE.Project

    Dim winFormUniqueName As String
    Dim userControlUniqueName As String
    Dim classLibUniqueName As String

    'Gather up the unique name of each project
    For Each project In DTE.Solution.Projects
        If (project.Name = "WinForm") Then
            winFormUniqueName = project.UniqueName
        ElseIf (project.Name = "UserControl") Then
            userControlUniqueName = project.UniqueName
        ElseIf (project.Name = "ClassLib") Then
            classLibUniqueName = project.UniqueName
        End If
    Next

    buildDependencies = DTE.Solution.SolutionBuild.BuildDependencies
    For Each buildDependency In buildDependencies
        If (buildDependency.Project.Name = "WinForm") Then
            buildDependency.RemoveAllProjects()
            'Add all projects except the WinForm
            ' project as a dependency:
            buildDependency.AddProject(userControlUniqueName)
            buildDependency.AddProject(classLibUniqueName)
        ElseIf (buildDependency.Project.Name = "UserControl") Then
            buildDependency.RemoveAllProjects()
            'Add a dependency to the ClassLib project:
            buildDependency.AddProject(classLibUniqueName)
        End If
    Next
End Sub
```

Manipulating Project Settings

Solution configurations are used to group together project configurations. Each project contains a number of configurations that control how the compiler should create the program code for that project. Because a project can have multiple project configurations associated with it, you can generate different versions of a program.

ConfigurationManager Object

You manage project configurations through the *ConfigurationManager* object, which has a collection of *Configuration* objects and lets you create new configurations. Configurations for a project are arranged in a grid pattern, with the configuration type, such as debug or release, along one axis of the grid and the platform on which the configuration will be built for on the other axis. The platforms that Visual Studio currently supports are Win32® for 32-bit Microsoft Windows® running on the x86 processor and Any CPU (the configuration name Any CPU is given to programs that will run on the .NET Framework), if the project is being compiled for the Microsoft .NET platform—including .NET applications for the desktop or smart device. Because projects can build only one platform type at a time, the second axis will always have one dimension.

Note Previous versions of Visual Studio used the name .NET for the platform name when compiling to MSIL bytecode. If you are upgrading an add-in or macro to Visual Studio, you will need to change your code to use the new name, Any CPU.

You can find a particular project configuration in several ways. The first way is to use the familiar *Item* method that's available on all collection objects. However, unlike most other *Item* methods on collection objects, the *ConfigurationManager.Item* method requires two parameters. The first parameter can be a numerical index and spans the entire grid of platforms and configurations. You can also use *Item* to directly locate a *Configuration* by passing the configuration name as the first parameter and the platform name as the second parameter. Suppose a Visual C++ project is open in Solution Explorer. To find the *Configuration* object for the Win32 debug build, you can use code such as the following:

```
Sub RetrieveDebugWin32Configuration()
    Dim config As Configuration
    Dim project As EnvDTE.Project
    project = DTE.Solution.Projects.Item(1)
    config = project.ConfigurationManager.Item("Debug", "Win32")
End Sub
```

Another way to retrieve specific configurations is to use the *ConfigurationManager.ConfigurationRow* and *ConfigurationManager.Platform* methods, which take the build type and the platform name, respectively. These methods return a collection of *Configuration* objects that you can iterate through to find a specific item. The *ConfigurationRow* method returns a list of all configurations with the passed name; the *Platform* method returns a list of all configurations belonging to a specific platform. These methods are most useful if you want to modify the settings of configurations that are closely related to one another, such as walking all the Win32 configurations of a Visual C++ project and enabling managed extensions, thus allowing your program to use the .NET Framework in C++ code. The following code sample does just that. After finding the Win32 configurations available to a

project, it retrieves the *Properties* object for that configuration and sets the *ManagedExtension* property to *true*, allowing the compiler to generate code that can work with the .NET Framework.

```
Sub SetManagedExtensionsProperty()
    Dim configManager As ConfigurationManager
    Dim configs As Configurations
    Dim config As Configuration
    Dim project As EnvDTE.Project
    project = DTE.Solution.Projects.Item(1)
    configManager = project.ConfigurationManager
    configs = configManager.Platform("Win32")
    For Each config In configs
        Dim prop As EnvDTE.Property
        prop = config.Properties.Item("ManagedExtensions")
        prop.Value = True
    Next
End Sub
```

You can create new configurations based on an existing configuration in the same way that you can create new solution configurations by copying an existing solution configuration. You create new project configurations by using the *ConfigurationManager.AddConfigurationRow* method. This method takes as its parameters the name of the new configuration and an existing configuration name, which is used as a template for creating the new configuration. *AddConfigurationRow* also accepts as an argument a *Boolean* value. This parameter, named *Propagate*, works in the same way as the *Propagate* parameter of the *SolutionConfigurations.Add* method, but in reverse. When the *SolutionConfigurations.Add* method is called with the *Propagate* parameter set to *true*, a copy of the solution configuration and all the project configurations it contains is made. If the *AddConfigurationRow* method is called with its *Propagate* parameter set to *true*, the currently active solution configuration is copied, its name is set to the name passed as the new project configuration, and the new solution configuration is modified to contain the newly created project configuration.

Note The *ConfigurationManager* object contains the method *AddPlatform*, which works much the same as the *AddConfigurationRow* method but adds a platform row to the build type configuration grid. If you call this method for any of the current versions of the Microsoft-language products, an exception will be generated because new platforms can't be added for these project types. This doesn't mean that this method won't work for third-party programming language projects or future versions of Microsoft programming languages.

Most project types support only one platform type, but some projects, such as setup projects, are not associated with any platform—what is built is platform-agnostic. A setup project doesn't care whether its contents are intended for Win32 or .NET platforms; its role is to contain files to be installed onto the user's computer, so a platform is not a consideration when you build a setup project. Because the build type configuration grid

can't be one-dimensional, a pseudoplatform is generated for these project types, and its name is set to <*N/A*>.

Project Configuration Properties

Project configurations differ in the property values that are set. For example, one difference between the debug and release configurations is that the debug configuration doesn't optimize the code, which makes debugging easier to perform, and optimization is turned on for the release configuration to make the code run faster. Such properties are set through the object returned by calling the *Configuration.Properties* property. As you saw earlier in the *SetManagedExtensionsProperty* macro example, this property returns an *EnvDTE.Properties* object—the same object that is used throughout Visual Studio to set property values on various objects. The following macro retrieves the debug and release configurations for a project, reads the *Boolean Optimize* configuration property, negates it, and then stores it back into the configuration. This means that the *Optimize* property is inverted for all these configurations.

```
Sub SwapOptimizationSettings()
    Dim project As EnvDTE.Project
    Dim configManager As EnvDTE.ConfigurationManager
    Dim configs As EnvDTE.Configurations
    Dim config As EnvDTE.Configuration
    Dim props As EnvDTE.Properties

    'Find the ConfigurationManager for the project:
    project = DTE.Solution.Projects.Item(1)
    configManager = project.ConfigurationManager

    'Get the debug configuration manager
    configs = configManager.ConfigurationRow("Debug")
    'Walk each configuration in the debug configuration row
    For Each config In configs
        Dim optimize As Boolean
        'Get the Optimize property for the configuration
        props = config.Properties
        optimize = props.Item("Optimize").Value
        'Negate the value
        props.Item("Optimize").Value = Not optimize
    Next

    'Repeat for the release configuration
    configs = configManager.ConfigurationRow("Release")
    For Each config In configs
        Dim optimize As Boolean
        'Get the Optimize property for the configuration
        props = config.Properties
        optimize = props.Item("Optimize").Value
        'Negate the value
        props.Item("Optimize").Value = Not optimize
    Next

End Sub
```

Build Events

As each stage of a build is performed, Visual Studio fires an event that can be captured by a macro or add-in, allowing custom code to be run. Four events are defined. Here are their signatures:

```
void OnBuildBegin(EnvDTE.vsBuildScope Scope, EnvDTE.vsBuildAction Action);
void OnBuildProjConfigBegin(string Project, string ProjectConfig,
    string Platform, string SolutionConfig);
void OnBuildProjConfigDone(string Project, string ProjectConfig,
    string Platform, string SolutionConfig, bool Success);
void OnBuildDone(EnvDTE.vsBuildScope Scope, EnvDTE.vsBuildAction Action);
```

These event handlers have the following meanings:

- ***OnBuildBegin*** This event is fired just before a build is started. Two arguments are passed to the handler of this event. The first argument is an enumeration of type *EnvDTE.vsBuildScope*, which can be either *vsBuildScopeBatch* (if you chose to start a batch build of one or more projects), *vsBuildScopeProject* (if you selected a single project to build by right-clicking a project and choosing Build), or *vsBuildScopeSolution* (if you chose the active solution configuration to build). The second argument is of type *EnvDTE.vsBuildAction* and can be either *vsBuildActionBuild* (if the project or solution configuration is to be compiled), *vsBuildActionClean* (if the project or solution configuration's build output is to be deleted from disk), *vsBuildActionDeploy* (if the project or solution configuration is to be deployed to its target), or *vsBuildActionRebuildAll* (if the project or solution configuration is to be rebuilt, even if the project's dependencies do not warrant a rebuild).
- ***OnBuildProjConfigBegin*** This event is fired when a project's configuration starts to be built. It is passed four arguments, each of type *string*. The first argument is the unique name of the project being built, the second is the name of the configuration being built, the third is the name of the platform being built, and last is the name of the solution configuration being built.
- ***OnBuildProjConfigDone*** This event handler is fired after a project configuration has been built. It is passed the same arguments as the *OnBuildProjConfigBegin* event, with the addition of a *Boolean* value that signals whether the configuration was built successfully (*true*) or failed to build (*false*).
- ***OnBuildDone*** This event is fired after all build steps have been completed, whether successfully or unsuccessfully.

Among the samples that accompany this book is one called BuildEvents, which demonstrates connecting to each of the build events. As each event handler is called, the information passed to that event handler is displayed within the output window, which contains information about the arguments that were passed to each handler. For example, if we were to create a solution containing two projects, ClassLibrary1 and ClassLibrary2, load

the sample add-in, and perform a build on the solution by choosing Build | Build Solution, the following information would be displayed:

```
OnBuildBegin
    Scope: vsBuildScopeSolution
    Action: vsBuildActionBuild

OnBuildProjConfigBegin
    Project: ClassLibrary1.csproj
    Platform: Any CPU
    Solution Configuration: Debug

OnBuildProjConfigDone
    Project: ClassLibrary1.csproj
    Platform: Any CPU
    Solution Configuration: Debug
    Success: True

OnBuildProjConfigBegin
    Project: ..\ClassLibrary2\ClassLibrary2.csproj
    Platform: Any CPU
    Solution Configuration: Debug

OnBuildProjConfigDone
    Project: ..\ClassLibrary2\ClassLibrary2.csproj
    Platform: Any CPU
    Solution Configuration: Debug
    Success: True

OnBuildDone
    Scope: vsBuildScopeSolution
    Action: vsBuildActionBuild
```

This output outlines the steps performed to build this two-solution project. It starts with a call to the *OnBuildBegin* event handler and then builds each project configuration contained within the solution configuration, one after another, with the *OnBuildDone* event handler being fired to signal that the build process has been completed. With Visual Studio, the *OnBuildProjConfigBegin* and *OnBuildProjConfigEnd* events are fired one after another, with no other build events fired between them. However, a macro or add-in should not take advantage of this order of events if you plan to port this code to a future version of Visual Studio because future versions might take advantage of multiprocessor computers, building one project configuration on one processor and another project configuration on another processor. If a macro or an add-in were to rely on this order of events, the code might not work properly.

Persisting Solution and Project Information Across IDE Sessions

At times, your add-in or macro might need to save some data that should be carried along with the solution or project file. The object model supports saving information into these files with the *EnvDTE.Globals* object. You can find this object by calling the *Globals* property of both of these objects:

```
Sub SolutionGlobals()
    Dim globals As EnvDTE.Globals
    globals = DTE.Solution.Globals
End Sub

Sub ProjectGlobals()
    Dim globals As EnvDTE.Globals
    globals = DTE.Solution.Projects.Item(1).Globals
End Sub
```

The *Globals* object of the *Solution* and *Project* objects works in much the same way as the *Globals* object found on the DTE object, with a few minor differences. First, if a macro or an add-in stores data into the solution or project file, even if the *VariablePersists* flag is set for that variable, the data might not be written into the solution or project file. This is because making a change to a variable causes the project or solution file to be put into a modified state. If you close the solution or project file but do not choose to save the modified files, the data won't be written into that file. Second, unlike the *EnvDTE.Globals* object on the DTE object, which can store data in a wide variety of formats, data stored into a solution or project file can be stored only in string format. This is because project and solution files are text-based, so any data stored into these files must also be in a text format. This doesn't mean that nonstring data can't be stored into the solution or project *Globals* object. It just means that when the data is to be written into the solution or project files, an attempt will be made to convert the data into a string. If that fails, the data won't be stored. Also, because the data is converted into a string when it is stored into the solution or project files, when the *Globals* object is restored from the solution or project file, this data will also be in a string format. It is up to the macro or add-in code to properly determine which format the data is in.

A good use of the *Globals* object is to keep track of the number of times you build a project. I like to count the number of times I build a project. Not that this number has any significance, but it is just an interesting fact. The following macro sample is an implementation of the *OnBuildDone* event. As each *OnBuildDone* event is fired, the sample checks for the existence of the *BuildCounter* variable within the solution *Globals* object. If this value exists, it is incremented and stored back into the *Globals* object. If this value doesn't exist, the value *1* is stored. The code for the *OnBuildDone* event is shown here:

```
Private Sub BuildEvents_OnBuildDone(ByVal Scope As _
    EnvDTE.vsBuildScope, ByVal Action As EnvDTE.vsBuildAction) _
    Handles BuildEvents.OnBuildDone
    'Increment the build counter by storing a value in the
    ' solution file through the Globals object:
    Dim globals As Globals
    Dim int32 As System.Int32
    globals = DTE.Solution.Globals
    If (globals.VariableExists("BuildCounter")) Then
        'A counter has been set, increment it:
        int32 = System.Int32.Parse _
           (globals.VariableValue("BuildCounter").ToString())
        int32 = int32 + 1
```

```
            globals.VariableValue("BuildCounter") = int32.ToString()
            globals.VariablePersists("BuildCounter") = True
        Else
            'The variable has never been set, seed the counter:
            globals.VariableValue("BuildCounter") = 1.ToString()
            globals.VariablePersists("BuildCounter") = True
        End If
    End Sub
```

Looking Ahead

In this chapter, we looked at how the pieces of the object model fit together to programmatically manage the many project types that can be loaded into a solution. In the next chapter, we will see how to program the user interface elements of Visual Studio, such as the many different tool and document windows.

Chapter 9

Programming the Visual Studio User Interface

In this chapter:

Microsoft® Visual Studio® is made up of many different windows that show data to the user, including the Task List, Solution Explorer, and the Microsoft Windows® Forms designer. You can manipulate these windows not only by using the mouse and keyboard but also through the object model by using a macro or an add-in. In this chapter, we'll discuss the many objects you can program in the user interface of Visual Studio.

Window Basics

The user interface for each window in Visual Studio is different from that of other windows, but they all share a few basic methods and properties. Let's look at the common parts of the object model.

The Windows Collection

Visual Studio contains a number of tool and document windows that you can access through the automation model. Each of these windows is represented in the object model by a *Window* object and can be found in the *Windows* collection, which is accessible through the *DTE.Windows* property.

You can retrieve a *Window* object from the *Windows* collection in a number of ways. One way is to use the enumerator to walk the list of all available windows, as shown here:

```
Sub EnumWindows()
    Dim window As EnvDTE.Window
    For Each window In DTE.Windows
        MsgBox(window.Caption)
    Next
End Sub
```

Or you can use the numerical indexing method:

```
Sub EnumWindows2()
    Dim window As EnvDTE.Window
    Dim i As Integer
    For i = 1 To DTE.Windows.Count
        MsgBox(DTE.Windows.Item(1).Caption)
    Next
End Sub
```

However, using these formats for finding a window isn't optimal because you usually want to find one specific window, and looking at all the windows to find it is a waste of CPU cycles. The numerical indexing method isn't always best because the position of a window from one instance of Visual Studio to the next might change, so you can't rely on using an index to return a specific *Window* object. In fact, you have no guarantee that calling the *Item* method twice in a row by using a numerical index will return the same *EnvDTE.Window* object because new windows might be created in between calls to this method. In addition, the numerical indexing method might not find all the available windows. For example, creating a tool window can be an expensive operation. To increase performance, Visual Studio won't create a tool window until one is specifically asked for, and, because the numerical indexing method looks only for windows that have been created, a particular tool window might not be found.

A simple experiment shows how iterating through the list of all tool windows slows down your code if all tool windows haven't been created. By default, the Server Explorer tool window is docked and hidden on the left side of the Visual Studio main window. If you move the mouse pointer over the icon for this window, the Server Explorer window appears. If this window hasn't yet been shown for that instance of Visual Studio, you'll see a delay of a couple seconds while the window is created before being shown for the first time. If you run the *EnumWindows2* macro and some of the *Window* objects need to be created, creating those windows will consume a lot of processor time, causing the macro to run very slowly.

Another way to find a window is to index the *Windows* collection by using the name of the window. The following macro demonstrates this approach; it uses the name of the Task List tool window to find the *Window* object for the Task List.

```
Sub FindTaskListWindow()
    Dim objWindow As EnvDTE.Window
    objWindow = DTE.Windows.Item("Task List")
End Sub
```

This is also not the best way of finding a particular *Window* object, as this example clearly shows. During a search for a window, the string passed to the *Windows.Item* method is compared with the title of each window until a window with a matching title is found. If you right-click on the Task List and choose Show Tasks | Comment, the title of this window becomes "Task List – X Comment tasks shown (filtered)," where *X* is a number. Because the string *Task List* passed to the *Item* method doesn't exactly match the title of the Task List window, the code *Windows.Item("Task List")* won't find the *Window* object. This isn't to say that you can't use the title indexing method in some situations. Some windows, such as the Properties window or Object Browser window, have names that don't change (unless the user is using a different language), and you can find such windows by using the window title as the index. Another reason why passing the title of a window isn't the best choice for the *Item* method is because, just as in the case of a numerical index, if the tool window hasn't been created, the *Window* object won't be found.

The best way to find a *Window* object is to use an index that is unique and independent of both the position within the *Windows* collection and the title of the window. Each tool window has a constant globally unique identifier (GUID) assigned to it; you can pass this GUID to the *Item* method to find the window you need. Because a GUID might be hard to remember, most of the tool windows that Visual Studio can create have constants defined that are easier to remember and recognize. These constants all start with the prefix *vsWindowKind* and are static (shared if you're using the Visual Basic language) members of either the *EnvDTE.Constants* class or the *EnvDTE80.WindowKinds* class. The following macro finds the Task List tool window:

```
Sub FindTaskListWindow2()
    Dim objWindow As EnvDTE.Window
    objWindow = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindTaskList)
End Sub
```

Because the GUID is unique to a specific tool window and doesn't change over time, you don't need to worry about either the caption of a window or its position within the *EnvDTE.Windows* collection changing. One other benefit of using the GUID is that even if the window you're searching for hasn't yet been created, Visual Studio is advanced enough to create the tool window when it's requested.

You might occasionally run across a window that doesn't have a constant GUID defined for it. The Source Control Explorer window is an example. When you need to find such a window, you can use the GUID in the form of a string in place of one of the predefined constants, as shown in the following example, which retrieves the *Window* object for the Source Control Explorer window:

```
Sub FindTheSourceControlExplorerWindow()
    Dim window As EnvDTE.Window
    window = DTE.Windows.Item("{99B8FA2F-AB90-4F57-9C32-949F146F1914}")
End Sub
```

You can find the GUID that can be passed to the *Item* method by using the *ObjectKind* property. The following macro takes this approach to display the GUID for the Favorites window:

```
Sub FindTheSourceControlExplorerWindow2()
    Dim window As EnvDTE.Window
    'You should show the Source Control Explorer window
    ' before calling this code!
    window = DTE.Windows.Item("Source Control Explorer")
    MsgBox(window.ObjectKind)
End Sub
```

When you run this macro, the GUID for the Source Control Explorer window is displayed in a message box. You can then define a constant set to this GUID, and use this constant in any code that needs to find this window. This is how we found the GUID for the *FindTheSourceControlExplorerWindow* macro.

Using the *Object* Property

Many windows in Visual Studio have an object model that you can use to manipulate the data contained in that window. You can find these window-specific objects by using the *Object* property of the *Window* object. For example, calling the *Object* property of the *Window* object for the Task List window returns the *TaskList* object, which allows you to enumerate, add, remove, and change properties of task items in the Task List window. The following macro retrieves the *TaskList* object:

```
Sub GetTaskListObject()
    Dim window As EnvDTE.Window
    Dim taskList As EnvDTE.TaskList
    window = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindTaskList)
    taskList = CType(window.Object, EnvDTE.TaskList)
End Sub
```

A number of types are available as the programmable object for the different windows, not just the *TaskList* object, as shown in the macro. Table 9-1 lists the GUID constant you pass to the *Item* method to find a *Window* object, as well as the programmable object for that window.

Table 9-1 Windows and Their Programmable Objects

Window	GUID Constant	Object Type
Command Window	*vsWindowKindCommandWindow*	*EnvDTE.CommandWindow*
Macro Explorer	*vsWindowKindMacroExplorer*	*EnvDTE.UIHierarchy*
Output window	*vsWindowKindOutput*	*EnvDTE.OutputWindow*
Server Explorer	*vsWindowKindServerExplorer*	*EnvDTE.UIHierarchy*
Solution Explorer	*vsWindowKindSolutionExplorer*	*EnvDTE.UIHierarchy*
Error List	*vsWindowKindErrorList*	*EnvDTE80.ErrorList*

Table 9-1 Windows and Their Programmable Objects

Window	GUID Constant	Object Type
Task List	*vsWindowKindTaskList*	*EnvDTE.TaskList*
Toolbox	*vsWindowKindToolbox*	*EnvDTE.ToolBox*
Web browser window	*vsWindowKindWebBrowser*	*SHDocVw.WebBrowser*
Text editor	<None>	*EnvDTE.TextWindow*
Forms designer	<None>	*System.ComponentModel.Design.IDesignerHost*
HTML designer	<None>	*EnvDTE.HTMLWindow*

Not only do some of the tool windows in Visual Studio have an object model, but a couple of the document windows have an object model, as well. The *Window.Object* property of the text editor, .NET Forms designer, and HTML designer windows returns an object appropriate for programming that window object. The object for programming the .NET Forms designer windows is discussed later in this chapter; the objects for programming the text editor and HTML editor windows are discussed in Chapter 10.

Shortcuts to Common Tool Windows

Although acquiring the specific object behind a tool window is not too terribly complicated, Visual Studio makes it easy to get to some of the most common tool window objects by using the *ToolWindows* object. This object gives you direct access to the object behind the Command Window, the Output window, Solution Explorer, Error List, Task List, and Toolbox. The *ToolWindows* object can be found on the *DTE2* object with code such as this very simple macro:

```
Sub FindTaskList()
    Dim taskList As TaskList

    taskList = CType(DTE, DTE2).ToolWindows.TaskList
End Sub
```

The *ToolWindows* object also gives you quick access to the specific object of other tool windows with the *GetToolWindow* property. This property accepts as a parameter the GUID for a tool window—the same GUID you would pass to the *Windows.Item* method, and returns the same value returned from a window's *Object* property, meaning that this line of code:

```
taskList = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindTaskList).Object
```

is equivalent to this line of code:

```
taskList = CType(DTE, DTE2).ToolWindows.GetToolWindow _
(EnvDTE.Constants.vsWindowKindTaskList)
```

The Main Window

Each tool and document window in Visual Studio has a *Window* object available. However, Visual Studio is also a window, so it's only fair that a *Window* object be available for that window, as well. Rather than indexing the *EnvDTE.Windows* collection to find this *Window* object, you use the *MainWindow* property of the *DTE* object:

```
Sub FindTheMainWindow()
    Dim mainWindow As EnvDTE.Window
    mainWindow = DTE.MainWindow
End Sub
```

When you work with the *Window* object for the Visual Studio main window, a few methods and properties don't work as they do when you work with tool or document *Window* objects. The differences between tool and document *Window* objects and the *Window* object for the main window are as follows:

- The *Document*, *Selection*, *Object*, *ProjectItem*, and *Project* properties return *null* if you're using Microsoft Visual C#® or Visual J#®, and they return *Nothing* if you're using Microsoft Visual Basic®.

  The set versions of the *Caption* and *Linkable* properties generate an exception if called.

- *IsFloating* and *AutoHides* generate an exception if you call the get or set versions of these properties.

- The *Close* method generates an exception if called.

Whereas a number of methods and properties don't work on the *Window* object for the main window, one property is available only for the main window. If an add-in or a macro needs to display a dialog box, you should supply a parent window when the dialog box is shown to correctly manage focus and set the "modalness" of the new window. You can use the main Visual Studio window as the parent window by calling the *Window.HWnd* property. This property returns a handle to a window—a Windows platform SDK *HWND* data type. This property is hidden, so when you develop your add-in or macro, it doesn't appear within statement completion. Because the .NET Framework can't use *HWND* values as a parent, this handle must first be wrapped by a class that implements an interface that the .NET library can accept as a parent. You can implement this interface, *System.Windows.Forms.IWin32Window*, on your add-in class or on a separate class within a macro project. The *IWin32Window* interface has one property named *Handle*; this property returns a *System.IntPtr*, which contains the handle to a parent window and, in this case, is the value returned from the *Window.HWnd* property. When it's time to show a form by using the *Form.ShowDialog* method, you can pass the class that implements the *IWin32Window* as an argument to this method.

To implement *IWin32Window* for an add-in, you must first add it to the interface list for your add-in, as shown here:

```
public class Connect : Object, Extensibility.IDTExtensibility2,
System.Windows.Forms.IWin32Window
```

Next, you add the implementation of the *Handle* property:

```
//Implementation of the IWin32Window.Handle property:
public System.IntPtr Handle
{
    get
    {
        return new System.IntPtr (applicationObject.MainWindow.HWnd);
    }
}
```

Finally, you can display a form (assuming that a form class named *Form1* exists within an add-in project) by using code such as this:

```
Form1 form1 = new Form1();
form1.ShowDialog(this);
```

Implementing this interface within a macro is even easier; the macro samples project that is installed with Visual Studio already contains the code for a class that implements this interface. Located in the Utilities module of the Samples project, this class, named *WinWrapper*, can be instantiated and passed to any code that requires a parent window, such as the standard Open File dialog box:

```
Sub ShowFileOpenDialog()
    Dim openFile As New OpenFileDialog
    openFile.ShowDialog(New WinWrapper)
End Sub
```

All you do is copy the *WinWrapper* class into your macro project, and it's ready to use.

Explorer Windows and the *UIHierarchy* Object

User interface hierarchy (or UI hierarchy) windows are tool windows that use a tree-like structure to display their data. Examples include the Solution Explorer, Server Explorer, and Macro Explorer windows. The *UIHierarchy* object and its associated objects, *UIHierarchyItems* and *UIHierarchyItem,* are so named because they represent a hierarchy of objects displayed in a tool window. The *UIHierarchy* object is used extensively by the macro recorder, allowing it to record the correct code to modify the selection within a UI hierarchy window; you can also use the *UIHierarchy* object as a valuable source of information about what is contained within these tool windows.

The *UIHierarchy* Object Tree

The *UIHierarchy, UIHierarchyItems,* and *UIHierarchyItem* objects work recursively. The *UIHierarchy* object is used to find the *UIHierarchyItems* collection, which contains all the root items of the tree within a UI hierarchy window. Each root tree item is represented by a *UIHierarchyItem* object within the *UIHierarchyItems* collection, and, because all of these tree items can themselves contain subitems, the *UIHierarchyItem.UIHierarchyItems* property

returns a *UIHierarchyItems* collection. This pattern of tree nodes returning a collection of other nodes continues until that branch of the tree ends. The following macro uses the *UIHierarchy* object to find and display the name of the top-level node of Macro Explorer:

```
Sub GetTopLevelUIHierItems()
    Dim macroExplWin As Window
    Dim uiHierarchy As EnvDTE.UIHierarchy
    Dim uiHierarchyItems As EnvDTE.UIHierarchyItems
    'Find the macro explorer window, and the UIHierarchy
    ' object for this window:
    macroExplWin = DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)
    uiHierarchy = macroExplWin.Object
    'Get the top level collection of items:
    uiHierarchyItems = uiHierarchy.UIHierarchyItems
    'Display the name of the first node in this collection:
    MsgBox(uiHierarchyItems.Item(1).Name)
End Sub
```

Here, Macro Explorer's *UIHierarchy* object is found and the collection of *UIHierarchyItems* is retrieved. The name displayed is that of the first item in the collection, which in this case is *Macros* because the top-level node in Macro Explorer is always the Macros node.

Continuing with our example, the Macros node in the Macro Explorer window contains a number of macro projects. Because this node can have subitems, it is a container of *UIHierarchyItem* objects, so the *UIHierarchyItem.UIHierarchyItems* property returns a collection object. This *UIHierarchyItems* collection contains a list of all the macro projects, and if we modify the earlier macro, we can walk the list of the macro projects:

```
Sub WalkMacroProjects()
    Dim macroExplWin As Window
    Dim uiHierarchy As EnvDTE.UIHierarchy
    Dim uiHierarchyItems As EnvDTE.UIHierarchyItems
    Dim uiHierarchyItem As EnvDTE.UIHierarchyItem
    Dim uiHierarchyItem2 As EnvDTE.UIHierarchyItem
    'Find the Macro Explorer window, and the UIHierarchy
    ' object for this window:
    macroExplWin = DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)
    uiHierarchy = macroExplWin.Object
    'Get the first node in this collection, the Macros node:
    uiHierarchyItem = uiHierarchy.UIHierarchyItems.Item(1)
    'Walk all the items in this collection, which is
    ' the list of macro projects:
    For Each uiHierarchyItem2 In uiHierarchyItem.UIHierarchyItems
        MsgBox(uiHierarchyItem2.Name)
    Next
End Sub
```

These sample macros show how to walk the hierarchy shown in the Macro Explorer window. To use this code to look at what is contained in the Solution Explorer and Server Explorer windows, you can simply change the value passed to the *Windows.Item* method to *Constants.vsWindowKindSolutionExplorer* or *Constants.vsWindowKindServerExplorer*.

Note Do the *UIHierarchy* objects seem familiar? Walking the *UIHierarchy*, *UIHierarchyItems*, and *UIHierarchyItem* objects to find an item in a UI hierarchy window is similar to using *ProjectItems* and *ProjectItem* to walk a project to find a project item. The reason for this similarity is that the *UIHierarchy* objects were designed to reflect how you would use the *ProjectItem* and *ProjectItems* objects.

The *UIHierarchy* Object

Finding a specific node within a UI hierarchy window can involve a great deal of code, especially if the desired node is nested more than two levels deep. Using the *UIHierarchy.GetItem* method, you can directly find a *UIHierarchyItem* object of a node rather than writing a lot of code to traverse the tree of nodes. For example, if you want to get to the *UIHierarchyItem* object of the *InsertDate* macro located in the VSEditor module of the Samples macro project, you can write code such as this:

```
Sub FindUIHierItemForInsertDateMacro()
    Dim macroExplWin As Window
    Dim uiHierarchy As EnvDTE.UIHierarchy
    Dim uiHierarchyItem As EnvDTE.UIHierarchyItem
    Dim uiHierarchyItems As EnvDTE.UIHierarchyItems
    macroExplWin = DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)
    uiHierarchy = macroExplWin.Object
    uiHierarchyItems = uiHierarchy.UIHierarchyItems
    uiHierarchyItem = uiHierarchyItems.Item("Macros")
    uiHierarchyItems = uiHierarchyItem.UIHierarchyItems
    uiHierarchyItem = uiHierarchyItems.Item("Samples")
    uiHierarchyItems = uiHierarchyItem.UIHierarchyItems
    uiHierarchyItem = uiHierarchyItems.Item("VSEditor")
    uiHierarchyItems = uiHierarchyItem.UIHierarchyItems
    uiHierarchyItem = uiHierarchyItems.Item("InsertDate")
    MsgBox(uiHierarchyItem.Name)
End Sub
```

This bit of code is quite verbose, however, and we can shorten it by using the *UIHierarchy.GetItem* method:

```
Sub FindUIHierItemForInsertDateMacro2()
    Dim macroExplWin As Window
    Dim uiHierarchy As EnvDTE.UIHierarchy
    Dim uiHierarchyItem As EnvDTE.UIHierarchyItem
    macroExplWin = DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)
    uiHierarchy = macroExplWin.Object
    uiHierarchyItem = _
        uiHierarchy.GetItem("Macros\Samples\VSEditor\InsertDate")
    MsgBox(uiHierarchyItem.Name)
End Sub
```

UIHierarchy.GetItem accepts a string, which is the path to an item that pinpoints a node within the hierarchy. This path is calculated by taking the names of each node in the branch to the tree node that you want to find, separated by the forward slash (\) character.

The *UIHierarchy.SelectedItems* property returns an array of *UIHierarchyItem* objects for items that are selected within the UI hierarchy tree. As do other arrays returned by the object model when you're using a language supported by .NET, this property returns an array of untyped objects—an array of *System.Object*.

```
Sub GetUIHierSelectedItems()
    Dim macroExplWin As Window
    Dim uiHierarchy As EnvDTE.UIHierarchy
    Dim selectedItems As Object()
    Dim uiHierarchyItem As EnvDTE.UIHierarchyItem
    macroExplWin = DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)
    uiHierarchy = macroExplWin.Object
    selectedItems = uiHierarchy.SelectedItems
    For Each uiHierarchyItem In selectedItems
        MsgBox(uiHierarchyItem.Name)
    Next
End Sub
```

To help the macro recorder record the movement of selections in a UI hierarchy window, the *UIHierarchy* object has two methods, *SelectUp* and *SelectDown*, that simulate the user selecting nodes within the tree. Both methods take two parameters as arguments. The first parameter is of type *EnvDTE.vsUISelectionType*, which denotes how nodes should be selected and closely reflects how the keyboard and mouse can be used to select particular nodes. *EnvDTE.vsUISelectionTypeSelect* selects a single node within the tree, causing any other selected node or nodes to lose their selection state. *EnvDTE.vsUISelectionTypeExtend* selects from the last selected node to the chosen node, much as if the user had clicked a node while holding down the Shift key. *EnvDTE.vsUISelectionTypeSetCaret* doesn't select a node—it moves the caret within the tree to the specified node. Lastly, *EnvDTE.vsUISelectionTypeToggle* swaps the selection state of a node, setting the selection if the node isn't selected or clearing the selection if it is selected. The second parameter of the *SelectUp* and *SelectDown* methods is a count parameter. By default, only one item is selected in either the up or down direction, but you can supply a different value so more than one node can be selected at one time.

The *UIHierarchy* object also has a method named *DoDefaultAction*. This method simulates the user pressing the Enter key with one or more nodes selected in the tree. For example, if a macro node is selected in Macro Explorer and the *UIHierarchy.DoDefaultAction* method is called, that macro runs.

The *UIHierarchyItems* Object

The *EnvDTE.UIHierarchyItems* object is a collection of *EnvDTE.UIHierarchyItem* objects and works as any other collection object in the Visual Studio object model. This object supports one property that is not part of the standard set of methods and properties of other collection objects: the *Expanded* property. This property is of type *Boolean* and returns *true* if the nodes underneath the *UIHierarchyItem* collection are shown in the user interface, and *false* otherwise. Setting this property to *True* has the same effect as the user clicking the plus symbol next to a tree view item; setting it to *False* is the same as the user clicking the minus symbol.

The *UIHierarchyItem* Object

The *EnvDTE.UIHierarchyItem*, being a collection item, supports the standard collection item methods and properties, such as *Collection* and *Name*. It also supports a method named *Select*. This method is similar to the *UIHierarchy.SelectUp* and *UIHierarchy.SelectDown* methods, except that it works on only one node at a time—the *UIHierarchyItem* that the *Select* method was called on. Because the *Select* method modifies only the current *UIHierarchyItem*, it doesn't accept a number of items to select.

Calling the *UIHierarchyItem.Object* property returns the extensibility object, if one is available, for that node. For example, when you're using Solution Explorer, you can retrieve an *EnvDTE.Project* or *EnvDTE.ProjectItem* object behind that node by using the *Object* property. The following code finds the *UIHierarchyItem* for the first project and second item within that project (the second item is searched for because the first item, when a .NET project is loaded, is the References node) and gets the *EnvDTE.Project* and *EnvDTE.ProjectItem* objects for those nodes:

```
Sub GetUIHierItemObject()
    Dim uihier As EnvDTE.UIHierarchy
    Dim uihierProj As EnvDTE.UIHierarchyItem
    Dim uihierProjItem As EnvDTE.UIHierarchyItem
    Dim project As EnvDTE.Project
    Dim projItem As EnvDTE.ProjectItem
    uihier = DTE.Windows.Item( _
        Constants.vsWindowKindSolutionExplorer).Object
    uihierProj = uihier.UIHierarchyItems.Item(1).UIHierarchyItems.Item(1)
    project = uihierProj.Object
    uihierProjItem = uihierProj.UIHierarchyItems.Item(2)
    projItem = uihierProjItem.Object
End Sub
```

The Toolbox Window

The Toolbox stores controls and code snippets that you can drag onto the Forms Designer window, text editor windows, and nearly anything else that can be a drag-and-drop target. The Toolbox is made up of a set of pages, or tabs, where items can be stored and grouped into related categories.

Tabs and Items

To find the Toolbox window, you can pass the constant *vsWindowKindToolbox* to the *Windows.Item* method, which returns a *Window* object. The *ToolBox* object is then found by calling the returned object's *Window.Object* property, as shown here:

```
Sub FindTheToolBox()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
```

```
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
End Sub
```

Because the Toolbox can contain more than one tab, a collection is available to enumerate all these tabs. You find this collection, the *ToolBoxTabs* object, by calling the *ToolBox.ToolBoxTabs* property. Using the *ToolBoxTabs* collection, you can enumerate each *ToolBoxTab* object in the Toolbox and even create new tabs to house components or text fragments of your choosing. To create a new tab, you use the *ToolBoxTabs.Add* method, which takes as an argument the name of the new tab to create and returns a *ToolBoxTab* object for the newly created tab. The following macro adds a new Toolbox tab:

```
Sub AddNewToolBoxTab()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    toolBox.ToolBoxTabs.Add("My commonly used items").Activate()
End Sub
```

This code creates a new tab called My Commonly Used Items, and the *Activate* method of the *ToolBoxTab* object makes sure it's the selected tab.

Not only is the Toolbox a collection of tabs, but each tab is also a collection of items. Each collection item is represented in the object model by a *ToolBoxItem* object and can be enumerated by using the *ToolBoxItems* object, which is found by calling the *ToolBoxTab.ToolBoxItems* property. You can walk the entire contents of the Toolbox by using the *EnumerateToolBoxContents* macro, shown here:

```
Sub EnumerateToolBoxContents()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    Dim toolBoxTab As ToolBoxTab
    Dim outputWindow As New _
        InsideVSNET.Utilities.OutputWindowPaneEx(DTE, "Toolbox contents")
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    For Each toolBoxTab In toolBox.ToolBoxTabs
        Dim toolBoxItem As ToolBoxItem
        outputWindow.WriteLine(toolBoxTab.Name)
        For Each toolBoxItem In toolBoxTab.ToolBoxItems
            outputWindow.WriteLine(vbTab + toolBoxItem.Name)
        Next
    Next
End Sub
```

Once you find a *ToolBoxItem* object, you'll see that you can't do much with it. You can call the *Select* method to make sure it's the active item in the Toolbox, you can remove the item by using the *Delete* method, and you can find the label that's displayed in the user interface by using the *Name* property. Although the object model of a *ToolBoxItem* is a functional dead end, the real power that the Toolbox object model offers you is the ability to create new items.

Adding Items to the Toolbox

The Toolbox can hold different types of objects, such as text, HTML, COM components, and .NET components. You can add your own items by using the *ToolBoxTab.Add* method, which takes three parameters. The first parameter, *Name*, is the display name of the item added; this string is the text that will be displayed within the Toolbox user interface. The second parameter, *Data*, defines the information stored in the Toolbox for the item. How this data is formatted depends on the third parameter, *Format*, which is of type *vsToolBoxItemFormat*.

The simplest data type that can be stored is raw text. The string passed to the *Data* parameter is copied verbatim into the Toolbox item, and when the text is dragged onto a window that supports drag-and-drop with a Clipboard format of type *text* (such as a text editor window), it is copied into that window. To add a text fragment, you can use code like this:

```
Sub AddTextToTheToolBox()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    Dim toolBoxTab As EnvDTE.ToolBoxTab
    Dim toolBoxItems As EnvDTE.ToolBoxItems
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    toolBoxTab = toolBox.ToolBoxTabs.Item("General")
    toolBoxItems = toolBoxTab.ToolBoxItems
    toolBoxItems.Add("My Text", "This is some text", _
        vsToolBoxItemFormat.vsToolBoxItemFormatText)
End Sub
```

This code starts by walking the object model and finding the General tab of the Toolbox. It ends by calling the *ToolBoxItems.Add* method and adding an item labeled *My Text* with the text *This is some text* that has the Clipboard format of type *text*.

Adding text in the HTML format is similar to adding plain text—the differences are that rather than passing raw text, you need to pass a fragment of HTML code, and the format of the data is marked as HTML by using *vsToolBoxItemFormatHTML*:

```
Sub AddHTMLToTheToolBox()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    Dim toolBoxTab As EnvDTE.ToolBoxTab
    Dim toolBoxItems As EnvDTE.ToolBoxItems
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    toolBoxTab = toolBox.ToolBoxTabs.Item("General")
    toolBoxItems = toolBoxTab.ToolBoxItems
    toolBoxItems.Add("My HTML", "<b>This is bold HTML</b>", _
        vsToolBoxItemFormat.vsToolBoxItemFormatHTML)
End Sub
```

After you run this code, a fragment of HTML is placed onto the Toolbox; if you drag that Toolbox item into the HTML designer, text will appear in bold style.

Note Remember that HTML is really just an application of XML that follows a particular schema. Because HTML is XML, you can also store XML fragments as HTML on the Toolbox. Visual Studio not only lets you drag-and-drop these HTML/XML fragments into an HTML document, but it also allows you to drag them into an XML document. In fact, a better name for the *vsToolBoxItemFormatHTML* value would have been *vsToolBoxItemFormatXML*.

Along with these two text formats, the Toolbox can also store ActiveX® controls, which can be dragged into HTML-designer or Win32® applications (such as a Microsoft Foundation Classes [MFC] dialog box) that support hosting ActiveX controls. To add an ActiveX control, supply the *vsToolBoxItemFormatGUID* data type. The format of the *Data* argument is the Class Identifier (CLSID) GUID of the ActiveX control or (despite the name of the format type) the ProgID of the control. The following macro adds two copies of the Windows Media® Player control to the Toolbox. The first one is added by using the CLSID of the control, and the second is added based on its ProgID:

```
Sub AddCOMObjectToTheToolBox()
    Dim toolBoxWindow As EnvDTE.Window
    Dim toolBox As EnvDTE.ToolBox
    Dim toolBoxTab As EnvDTE.ToolBoxTab
    Dim toolBoxItems As EnvDTE.ToolBoxItems
    toolBoxWindow = DTE.Windows.Item(Constants.vsWindowKindToolbox)
    toolBox = toolBoxWindow.Object
    toolBoxTab = toolBox.ToolBoxTabs.Item("General")
    toolBoxItems = toolBoxTab.ToolBoxItems
    toolBoxItems.Add("Name", "{22D6F312-B0F6-11D0-94AB- 0080C74C7E95}", _
        vsToolBoxItemFormat.vsToolBoxItemFormatGUID)
    toolBoxItems.Add("Name", "MediaPlayer.MediaPlayer.1", _
        vsToolBoxItemFormat.vsToolBoxItemFormatGUID)
End Sub
```

When you run this code, you'll notice that the *Name* parameter is ignored. This is because the Toolbox extracts the name from the control.

Note .NET User Controls can be added to the toolbox by using the *vsToolBoxItemFormat.vsToolBoxItemFormatDotNETComponent* enumerated value. However, this method has been deprecated in favor of using the Content Installer to install controls, as we discussed in Chapter 4. The Content Installer gives you much better control over where items are installed into the toolbox and makes installing and uninstalling controls much easier.

The Task List Window

As you saw earlier, the programmable object behind the Task List window is the *EnvDTE.TaskList* object. Using the *TaskList* object, you can add new task items to provide information to the user about work that needs to be performed, as well as examine tasks added by a compiler or other tool.

Task List Items

The *EnvDTE.TaskList* object lets you get to the items in the Task List window by calling the *TaskItems* property, which returns a *TaskItems* collection containing one item for each task item in the Task List window. You can view subsets of the items in the Task List window by filtering out items that don't belong to a particular grouping, or category, but items that are hidden because of this filtering will still have an item in the *EnvDTE.TaskItems* collection.

As with any other collection, you can index *EnvDTE.TaskItems* by its numerical position, which returns an *EnvDTE.TaskItem* object. You can use a number as an index to this collection, but it doesn't have a string format as an index.

Adding New Tasks

You can add new items to the Task List window to build a wide range of new tools. Examples of tools you can build that use the Task List window include:

- Code analysis tools that find common programming errors, letting you find a bug before your customer does. You can place details about these errors in the Task List window alongside compiler errors. These tools are sometimes called *Lint* tools.
- Scheduling tools that pull information from other software such as Microsoft Project and create task items to let programmers know when a specific portion of their work is due. When the check box next to a task item is selected, the corresponding item in Project is marked as completed.
- An add-in that searches through compiler errors and fixes as many as it can. Remember the last time you compiled a C# project, only to have errors generated because you forgot a semicolon? Wouldn't it be great to have a tool to fix this automatically?
- A macro that synchronizes your calendar in Microsoft Outlook® with the Visual Studio Task List window, reminding you, among other things, to pick up a gift on your way home from work for an anniversary or a birthday. (Such a tool can save you a lot of grief.)

You can build such tools because you can insert new task items into the Task List window by using the *TaskItems.Add* method. *TaskItems.Add* offers a great deal of flexibility in the elements that are displayed for new task items and how they're displayed. As a result, this method has one of the most complex argument signatures of all the methods in Visual Studio:

```
public EnvDTE.TaskItem Add(string Category,
    string SubCategory,
    string Description,
    EnvDTE.vsTaskPriority Priority = vsTaskPriorityMedium,
    object Icon,
    bool Checkable = false,
    string File = "",
    int Line = -1,
    bool CanUserDelete = true,
    bool FlushItem = true)
```

You can use the sample add-in AddTaskListItems to see the output generated by the many combinations of these parameters. We'll look at each parameter in turn over the next few sections.

Category and *SubCategory*

All tasks, whether they're created by the automation object model or by Visual Studio 2005 itself, belong to a category. *Categories* are used simply to group tasks and relate them to one another. Common category types are compile errors, user tasks, and shortcuts. You can create new category groups by using the *Category* parameter of the *Add* method. When you call the method, the list of currently known categories is searched for a category with a name that matches this argument. If one is not found, a new category is added and the new task item is added to this category. If a category with a matching name is found, the new task is added to that existing category group.

Visual Studio doesn't currently use the *SubCategory* argument of the *Add* method; your add-in or macro can leave it blank, or use it for internal bookkeeping.

Description

The description of a task appears in the Description column of the Task List window, and the *Description* argument of the *Add* method sets this column. This parameter of the *Add* method and the *Category* and *SubCategory* parameters are the only required parameters. Ignoring the optional parameters for now, we'll create our first Task List item by using the following macro code:

```
Sub TLAddItems()
    Dim taskList As EnvDTE.TaskList
    taskList = DTE.Windows.Item(Constants.vsWindowKindTaskList).Object
    taskList.TaskItems.Add("Category", "", "Description2")
    taskList.TaskItems.Add("Category", "", "Description1")
End Sub
```

Priority

The next argument you can pass to the *Add* method is the *Priority* argument. This argument is optional when you use the Visual Basic programming language, but if it's supplied, an icon appears in the first column of the Task List—the priority column—to remind the user of the importance of completing that task. A high-priority task has a red exclamation point next to it, a low-priority task has a blue downward-pointing arrow, and a medium-priority task has no priority icon. The following macro adds new task items to the Task List, each with a different priority.

```
Sub TLAddItemsPriority()
    Dim taskList As EnvDTE.TaskList
    taskList = DTE.Windows.Item(Constants.vsWindowKindTaskList).Object
    taskList.TaskItems.Add("Category", "", _
        Description1", vsTaskPriority.vsTaskPriorityHigh)
```

```
    taskList.TaskItems.Add("Category", "", _
       "Description2", vsTaskPriority.vsTaskPriorityLow)
    taskList.TaskItems.Add("Category", "", _
       "Description3", vsTaskPriority.vsTaskPriorityMedium)
End Sub
```

Icon

The *Icon* parameter allows you to place an icon next to a newly added task item to identify that task item. The five predefined icons are described in Table 9-2.

Table 9-2 Predefined Icons for Task List Items

Icon Image	Constant
	*vsTaskIcon.vsTaskIconComment*
	*vsTaskIcon.vsTaskIconUser*
	vsTaskIcon.vsTaskIconSquiggle
	*vsTaskIcon.vsTaskIconShortcut*
	*vsTaskIcon.vsTaskIconCompile*

Left out of this table is the default icon, *vsTaskIconNone*–a blank icon–which appears (or, in this case, does not appear) if this parameter is not specified.

Note If you call the *TaskItems.Add* method and supply the value *vsTaskIconShortcut* as the icon, a shortcut in a file isn't created. The icon is used for display purposes only. This applies to the other values that can be passed as the *Icon* parameter; using *vsTaskIconCompile* doesn't create a compiler error, *vsTaskIconComment* doesn't add a comment to a source file, and so forth.

If these predefined images don't suit the task item you're creating, you can create your own image to display next to the task. You need a 16-by-16-pixel bitmap image with a color depth of either 16 colors or 16,777,215 (24-bit) colors. Any pixel in the image that has a background RGB color of 0,254,0 will bleed through the image, showing the color of the Task List window. The *Icon* paramater can be set to one of three formats in addition to the previously listed constants. You can pass the handle of a *System.Drawing.Bitmap* object, which is retrieved from the *GetHbitmap* method. You can also pass the path as a string to a bitmap file on disk. The final way is to load the bitmap into an *IPictureDisp* instance and then pass it as the *Icon* parameter. An *IPictureDisp* interface is the COM way of passing around a bitmap object. To create an *IPictureDisp* object in a .NET add-in, you must write a small amount of P/Invoke code to create this object type. (P/Invoke is the technology the .NET Framework uses to call unmanaged code from .NET programs.) The system DLL, oleaut32.dll, exports a method called *OleLoadPictureFile* that takes a path to a bitmap file,

which can be the bitmap to show in the Task List, and returns the necessary *IPictureDisp* object. Before you call the *OleLoadPictureFile* method, you must add some code that might seem magical to the class that implements your add-in:

```
[DllImport("oleaut32.dll",
    CharSet=System.Runtime.InteropServices.CharSet.Auto,
    SetLastError=true)]
internal extern static int OleLoadPictureFile(object fileName,
[MarshalAsAttribute(UnmanagedType.IDispatch)] ref object ipictrueDisp);
```

This code defines the method signature for code that's exported from the COM DLL OleAut32.dll, and, with this P/Invoke method declared, you can call the *OleLoadPictureFile* method with the file name of the custom bitmap:

```
object objIPictureDisp = null;
string filename = "C:\SomeImage.bmp";
int nret = OleLoadPictureFile(fileName, ref objIPictureDisp);
```

When this method call returns, the *objIPictureDisp* variable is set to an *IPictureDisp* object that can be passed as the *Icon* parameter of the *TaskItems.Add* method.

If you try to call the *Add* method from within a macro and pass as the *Icon* parameter an *IPictureDisp* object or the handle to a bitmap, an exception is generated. This happens because your macros run in a separate process. When a method or property is called on the Visual Studio object model, all data must be marshaled, or translated from the memory being used by the Macros editor program to the memory used by the Visual Studio program, across the process boundaries. However, objects such as *IPicture*, *IPictureDisp*, and *HBITMAP* can't be marshaled across processes, so if you try to create an *IPicture*, *IPictureDisp*, or *HBITMAP* and pass it to the *TaskItems.Add* method from a macro, an exception will be generated. This limits you to passing only a file path or one of the constants from a macro, but you can create and use a custom bitmap from within an add-in by using *IPictureDisp* or a handle because add-ins are loaded into the same process as Visual Studio.

Checkable

The *Checkable* parameter of *TaskListItems.Add* controls whether the check box appears next to a task item. If it's set to *true*, the check box is available; if it's set to *false*, the check box does not appear.

File and *Line*

File and *Line* are a string and an integer, respectively, that fill out the File and Line columns of the Task List. These can contain any values you want—they're not used in any way other than for information to display within the Task List. If the user later performs the default action on the task (either double-clicking or pressing the Enter key when the task item is selected), the file won't open and the caret won't be placed on the line specified in the *Line* argument unless you write a little extra code, which will be discussed shortly.

CanUserDelete

The *CanUserDelete* parameter controls whether the user can delete the task item by pressing the Delete key when the task is selected in the user interface. If this value is set to *false*, the user cannot delete the item, but you can still delete it through the object model by calling the *TaskItem.Delete* method.

FlushItem

The last parameter of *TaskListItems.Add* is a Boolean value called *FlushItem*. As each new item is inserted into the Task List, the Task List must be updated to show the new task. If you add a large number of tasks, redrawing the Task List each time an item is added will slow down your application's performance. If you pass a *false* value as the *FlushItem* argument, no updates are made to the Task List until either another task item is added that does an update or the method *TaskItems.ForceItemsToTaskList* is called.

The *TaskItem* Object

Once an item has been added to the Task List—whether it was created by using the *TaskItems.Add* object or created by Visual Studio itself and obtained by the *TaskItems* collection—you can use the *TaskItem* object's methods and properties to examine and modify the data displayed for that task item. The properties *Category*, *SubCategory*, *Checked, Description, FileName, Line,* and *Priority* each can be read programmatically to see what data is stored for those columns of the Task List. You can also set these properties as long as they're not read-only. Some task items that Visual Studio creates have their columns marked as read-only so they can't be modified. To test whether a particular column can be set, you can make a call to the *IsSettable* property, which accepts as a parameter the column within the Task List (a *vsTaskListColumn* enumeration value), and if the column can be modified, the *IsSettable* property returns true; otherwise, it returns false. For example, to change a task item's description value, you can write code such as this, which first verifies that the description can be changed:

```
Sub ModifyTaskDescription()
    Dim taskList As EnvDTE.TaskList
    Dim task As EnvDTE.TaskItem
    taskList = DTE.Windows.Item(Constants.vsWindowKindTaskList).Object
    task = taskList.TaskItems.Item(1)
    If (task.IsSettable(vsTaskListColumn.vsTaskListColumnDescription)) Then
        task.Description = "A new description"
    End If
End Sub
```

Delete deletes the item, if deletion is possible, from the list of items. As mentioned earlier, all items added through the object model can be deleted whether the *CanUserDelete* parameter is *true* or *false* when you call the *TaskItems.Add* method. Other task items can be deleted depending on who created them. For example, if the task item was added by the user clicking on the Create User Task button on the command bar of the Task List when the User Tasks category is selected, it can be deleted by using the object model. If the item was created by

IntelliSense®, an exception is generated when this method is called because the only way to remove the task item is to modify the source code that's causing the task item to appear.

AutoNavigate

The *TaskItem.Navigate* method simulates the user double-clicking or pressing the Enter key when the task has the focus. If the task was added by Visual Studio or by a compiler, or if the task is a shortcut task, this opens the target file and places the caret on the line specified by the task. By default, if the task was added through the automation model, no action is taken unless you write code to manually navigate to the proper file and line location by using the *TaskNavigated* event. However, you can also let Visual Studio handle opening the referenced document and line number through the *AutoNavigate* parameter of the *TaskItems2.Add2* method. *TaskItems2.Add2* looks very similar to the *TaskItems.Add* method, except it takes one additional parameter, *AutoNavigate*. If this parameter is set to *true*, when the user double-clicks the task item or calls *Navigate* on the *TaskItem* object, then the document path given in the *File* parameter is opened—or, if the document is already open, it is made active, and then the caret is placed at the beginning of the line indicated by the *Line* parameter. You should make sure that if you specify true to *AutoNavigate*, the file is a text document and not a binary file; otherwise, unpredictable results may occur.

Task List Events

As the user interacts with the Task List, events are fired to allow your add-in or macro to respond to those user interactions. Possibly the most important event of your add-in or macro that adds task list items is the *TaskNavigate* event. This event is fired when the user double-clicks on a Task List item, presses the Enter key when a task has the focus, or chooses Next Task or Previous Task from the Task List's shortcut menu. To capture this event, you can connect to the *TaskListEvents.TaskNavigated* event. This event is passed the *TaskItem* object of the item that the user wants to navigate to, plus a reference to a Boolean value called *NavigateHandled* that you can use to tell Visual Studio whether your code has handled the navigation of the task item. If the value *false* is passed back through the *NavigateHandled* argument and no one else handles the navigation of the task, Visual Studio plays a bell sound for the user.

Connecting to this event within a macro project is as simple as opening the *EnvironmentEvents* macro module, selecting the TaskListEvents item from the drop-down list at the top left of the editor window for this module, and then selecting the *TaskNavigated* event from the top-right drop-down list. Using this event and the arguments that are passed to it, you can write a macro event handler for the *NavigateHandled* event that opens the file (if specified) that the task item refers to and select the line in the source file that the task item points to. The code for this event handler would look like this:

```
Private Sub TaskListEvents_ TaskNavigated(ByVal TaskItem As EnvDTE.TaskItem, _
    ByRef NavigateHandled As Boolean) Handles TaskListEvents.TaskNavigated
    'If the file argument has been specified for this task...
```

```
    If (TaskItem.FileName <> "") Then
        Dim fileWindow As EnvDTE.Window
        Dim textWindow As EnvDTE.TextWindow
        Dim textPane As EnvDTE.TextPane

        'Then open the file, find the TextWindow and TextPane objects...
        fileWindow = DTE.ItemOperations.OpenFile(TaskItem.FileName, _
            EnvDTE.Constants.vsViewKindTextView)
        textWindow = CType(fileWindow.Object, EnvDTE.TextWindow)
        textPane = CType(textWindow.ActivePane, EnvDTE.TextPane)

        'Then move the caret to the correct line:
        textPane.Selection.MoveTo(TaskItem.Line, 1, False)
        textPane.Selection.SelectLine()
        NavigateHandled = True
    End If
End Sub
```

Connecting to this event within an add-in is almost as simple as connecting to it within a macro, but a little more code is needed. The first step is to declare a variable to connect the event handler to. In this case, we're connecting to Task List events, so we'll use the *EnvDTE.TaskListEvents* class:

```
private EnvDTE.TaskListEvents taskListEvents;
```

Next, you declare the event handler method, which must follow the method signature as declared in the Object Browser. You can also convert the macro code shown earlier into C# for an add-in:

```
public void TaskNavigated(EnvDTE.TaskItem taskItem,
    ref bool navigateHandled)
{
    //If the file argument has been specified for this task...
    if(taskItem.FileName != "")
    {
        EnvDTE.Window fileWindow;
        EnvDTE.TextWindow textWindow;
        EnvDTE.TextPane textPane;

        // Then open the file, find the TextWindow and TextPane objects...
        fileWindow = applicationObject.ItemOperations.OpenFile(
            taskItem.FileName, EnvDTE.Constants.vsViewKindTextView);
        textWindow = (EnvDTE.TextWindow)fileWindow.Object;
        textPane = (EnvDTE.TextPane)textWindow.ActivePane;

        //Then move the caret to the correct line:
        textPane.Selection.MoveTo(taskItem.Line, 1, false);
        textPane.Selection.SelectLine();
        navigateHandled = true;
    }
}
```

Finally, you must set the *taskListEvents* variable to an instance of a *TaskListEvents* object, which you find by calling the *Events.TaskListEvents* property. This property takes one

argument—a category that's used as a filter. If you pass the empty string as an argument, your event handler is called when any task item generates an event—whether the item was added by an add-in or macro or by Visual Studio itself. But if you specify a category for this argument—the same category string you can pass as the first argument to the *TaskItems.Add* method—only events for a task item that have this same category are sent to your event handler. This filtering mechanism can help cut down on the number of events that are fired, thereby increasing the performance of your code. Because we want our code to handle events for all task items, we'll pass the empty string to the *Events.TaskListEvents* property:

```
EnvDTE.Events events = applicationObject.Events;
taskListEvents = (EnvDTE.TaskListEvents)events.get_TaskListEvents("");
```

The last step is to associate the event object with the event handler. You do this by creating a new *EnvDTE._dispTaskListEvents_TaskNavigatedEventHandler* object and adding it to the *taskListEvents.TaskNavigated* collection of event handlers:

```
taskListEvents.TaskNavigated += new
    _dispTaskListEvents_TaskNavigatedEventHandler(this.TaskNavigated);
```

TaskNavigated isn't the only Task List event your code can capture. *TaskAdded* and *TaskRemoved* events are fired when a new task item is added or just before it's removed, respectively. The last event, *TaskModified*, is fired when one of the columns of the Task List is modified. For instance, the user can check or uncheck an item or change the priority or descriptive text for a task item. To let your code know when these tasks are changed, the *TaskModified* event is fired, passing the task item and the column that was modified.

Comment Tokens

Developers commonly leave portions of their code incomplete as they work, with the intention of finishing it later. This omitted code might include error-checking, some parameter validation, or notes to themselves to handle a few additional code paths. Of course, unless you specifically search through the code for these tokens, either by visually inspecting it or by using the text editor search tools, you might never revisit these notes and make the corrections. However, if you use a special notation, the Task List can find and report these notes for you automatically. When you open a source file, the file is scanned for these special tokens, and, if any are found, an entry is made in the Task List. The tokens in the source file have the format of a language comment marker followed by the comment token, the colon (:) character, and finally the note that is to appear within the Task List. For example, the comment *//TODO: Fix this later* creates a Task List item for Microsoft Visual C++ and C# with the description *Fix this later*, and the comment *'TODO: Fix this later'* does the same for a Visual Basic file.

The special tokens that the Task List searches for are defined in the Options dialog box, where you can add new tokens and remove or modify existing tokens. The default comment tokens are HACK, TODO, UNDONE, and UnresolvedMergeConflict.

You can add a new token by typing a token name in the Name box, selecting a priority, and then clicking the Add button. You can also add, remove, and modify these tokens by using the object model. To program these tokens, use the *Properties* collection. You'll find more details about the *Properties* collection later in this chapter—for now, we'll overlook the details of how to use the *Properties* collection and look only at how to change the tokens by using this object. The first step is to find the *CommentTokens* property by using code such as the following:

```
Sub GetTokensArray()
    Dim tokens As Object()
    Dim prop As EnvDTE.Property
    Dim props As EnvDTE.Properties
    props = DTE.Properties("Environment", "TaskList")
    prop = props.Item("CommentTokens")
    tokens = prop.Value
End Sub
```

The *CommentTokens* property returns an array of strings that have a special format, and when this macro is run, it finds all the available tokens in the format *TokenName*: *Priority*, where *TokenName* is what should appear after the comment notation for the given language and *Priority* is the numerical value of an item in the *EnvDTE.vsTaskPriority* enumeration. In the preceding macro, the string for the TODO token is *TODO:2* because the string to search for in the text editor is *TODO* and the priority that appears in the Task List for this token is *vsTaskPriorityMedium* (whose numerical value is 2).

Adding your own token to the list of tokens is a three-step process. Setting the list of tokens clears the current list (at least the tokens that are not read-only), so you need to preserve the known tokens so you don't overwrite the known tokens that the user might have created. First, you need to retrieve the list of current Task List tokens. Add your own token to the array of existing tokens, and then set the property with the expanded array. You can see this in the following macro, which adds a high-priority SECURITY token to the list of comment tokens:

```
Sub AddSecurityToken()
    Dim tokens As Object()
    Dim token As String
    Dim prop As EnvDTE.Property
    Dim props As EnvDTE.Properties
    'Find the property holding the known tokens
    props = DTE.Properties("Environment", "TaskList")
    prop = props.Item("CommentTokens")
    tokens = prop.Value
    Add one to the list of known tokens to hold
    ' the new SECURITY token
    ReDim Preserve tokens(tokens.Length)
    'Add the new token
    tokens(tokens.Length - 1) = "SECURITY:3"
    'Set the list of known tokens
    prop.Value = tokens
End Sub
```

To delete a token, you run similar code, but instead of adding an element to the array, you remove an element:

```
Sub RemoveSecurityToken()
    Dim tokens As Object()
    Dim newTokens As Object()
    Dim token As String
    Dim i As Integer = 0
    Dim found As Boolean = False
    Dim prop As EnvDTE.Property
    Dim props As EnvDTE.Properties
    props = DTE.Properties("Environment", "TaskList")
    prop = props.Item("CommentTokens")
    tokens = prop.Value
    'Don't want to shrink the array if
    '  the token is not available
    For Each token In tokens
        If token = "SECURITY:3" Then
            found = True
            Exit For
        End If
    Next
    'If the SECURITY token was not found, then
    ' there is nothing to remove so we can exit
    If found = False Then
        Exit Sub
    End If
    'Resize the newTokens array
    ReDim newTokens(tokens.Length - 2)
    'Copy the list of tokens into the newTokens array
    ' skipping the SECURITY token
    For Each token In tokens
        If token <> "SECURITY:3" Then
            newTokens(i) = token
            i = i + 1
        End If
    Next
    'Set the list of tokens
    prop.Value = newTokens
End Sub
```

If your add-in generates code to place in the text buffer and you want to insert a comment token that gives the user additional information about how to modify the code, you can use the *TaskList.DefaultCommentToken* property to find which token to insert. The following code creates a string containing a class, with the default comment token directing the user to where to insert code:

```
Sub InsertTLTokenCode()
    Dim classString As String
    Dim taskList As EnvDTE.TaskList
    taskList = DTE.Windows.Item(Constants.vsWindowKindTaskList).Object
    classString = "Public Class AClass" + Chr(13)
    classString = classString + Chr(9) + "'" + taskList.DefaultCommentToken
```

```
    classString = classString + ": Insert your code here" + Chr(13)
    classString = classString + "End Class"
End Sub
```

The Error List Window

The error list window looks like and acts very much like the task list, except this window is read-only, meaning you cannot add your own items to it—adding items is reserved only for compilers. When the code is compiled within Visual Studio, the output of the compilers are scanned, and one item is inserted into the error list for each error, warning, or message found. If you were to browse the objects, methods, and properties of the error list, you will find that many of the methods and properties on the *ErrorList*, *ErrorItems*, and *ErrorItem* objects work much like the methods and properties of the *TaskList*, *TaskItems*, and *TaskItem* objects.

The *ErrorList* object, the object representing the error list tool window in the user interface, can be found either by using the *ToolWindows.ErrorList* property or the *Windows.Item* method, passing in the *EnvDTE80.WindowKinds.vsWindowKindErrorList* constant, and then casting the *Window.Object* property into an *ErrorList* type:

```
    Sub FindErrorList()
        Dim errorList As ErrorList

        'Find the ErrorList object using the Windows.Item(...).Object style
        errorList = DTE.Windows.Item _
(EnvDTE80.WindowKinds.vsWindowKindErrorList).Object

        'Find the ErrorList object using the ToolWindows style
        errorList = CType(DTE, DTE2).ToolWindows.ErrorList
    End Sub
```

Once you have the *ErrorList* object, you can get to the *ErrorItems* object, and then to an *ErrorItem* object. The differences between these objects and their counterparts, *TaskList*, *TaskItems*, and *TaskItem*, are that, obviously, the name Task has been replaced with Error; in objects, methods, and properties, the *Add* method has been removed from the collection object; and the properties *ShowErrors*, *ShowWarnings*, and *ShowMessages* have been added to the item object. These properties, which take and return a Boolean value, allow you to control and inspect to determine whether errors, warnings, and messages are visible within the error list.

The Output Window

The Output window is where Visual Studio displays text information generated by tools such as compilers or the debugger. The Output window is also a perfect place for any tools you create that generate text information that might be useful to the user. In fact, throughout this book, the sample macros and add-ins use the class library *OutputWindowPaneEx* to display text in the Output window as these samples do their work.

The object behind the Output window is called *OutputWindow*, and you can find this object by using code such as this:

```
Sub FindOutputWindow()
    Dim window As EnvDTE.Window
    Dim outputWindow As EnvDTE.OutputWindow
    window = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindOutput)
    outputWindow = CType(window.Object, EnvDTE.OutputWindow)
End Sub
```

Output Window Panes

The user interface of the Output window consists of a number of view ports, or panes, each of which displays text. You can switch between these panes by selecting a pane by name from the drop-down list at the top of the Output window. You can enumerate the panes by using the *OutputWindowPanes* object, as shown here:

```
Sub EnumOutputWindowPanes()
    Dim window As EnvDTE.Window
    Dim outputWindow As EnvDTE.OutputWindow
    Dim outputWindowPanes As EnvDTE.OutputWindowPanes
    Dim outputWindowPane As EnvDTE.OutputWindowPane
    'Find the OutputWindow object
    window = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindOutput)
    outputWindow = CType(window.Object, EnvDTE.OutputWindow)
    'Retrieve the OutputWindowPanes object
    outputWindowPanes = outputWindow.OutputWindowPanes
    'Enumerate each OutputWindowPane
    For Each outputWindowPane In outputWindowPanes
        MsgBox(outputWindowPane.Name)
    Next
End Sub
```

You can also use the *OutputWindowPanes* object to create new panes. The method *Add* takes as its only argument the name of the new pane to create:

```
Sub CreateOutputWindowPane()
    Dim window As EnvDTE.Window
    Dim outputWindow As EnvDTE.OutputWindow
    Dim outputWindowPanes As EnvDTE.OutputWindowPanes
    Dim outputWindowPane As EnvDTE.OutputWindowPane
    'Find the OutputWindow object
    window = DTE.Windows.Item(EnvDTE.Constants.vsWindowKindOutput)
    outputWindow = CType(window.Object, EnvDTE.OutputWindow)
    'Retrieve the OutputWindowPanes object
    outputWindowPanes = outputWindow.OutputWindowPanes
    'Add a new pane:
    outputWindowPane = outputWindowPanes.Add("My New Pane")
End Sub
```

This macro creates a new output window pane named My New Pane that's ready to be filled with the text output of your add-in or macro code. You can inject text into this window by

using the *OutputWindowPane.OutputString* method, which takes a string that's appended to the end of other text in the appropriate pane. As strings are placed into the Output window pane, they're injected without a line break between them; this means that if a new line character needs to be placed between each string, you must write the code to do this. The following macro sample displays the contents of the folder containing the solution file that's currently open; as each file path is displayed in the Output window pane, a line break (or ASCII value 13) is inserted:

```
Sub DisplaySolutionDirectory()
    Dim files As String()
    Dim file As String
    Dim directoryOutputWindowPane As OutputWindowPane
    Dim fullName As String
    Dim outputWindow As OutputWindow
    outputWindow = DTE.Windows.Item(Constants.vsWindowKindOutput).Object

    'Find the folder the solution is in, as well as the files that are
    ' in that folder:
    fullName = System.IO.Path.GetDirectoryName(DTE.Solution.FullName)
    files = System.IO.Directory.GetFiles(fullName)

    'Try to find a "Solution Directory" pane, if one does not exist,
    ' create it:
    With outputWindow.OutputWindowPanes
        Try
            directoryOutputWindowPane = .Item("Solution Directory")
            'Show the pane:
            directoryOutputWindowPane.Activate()
        Catch
            directoryOutputWindowPane = .Add("Solution Directory")
        End Try
    End With
    'Clear the pane:
    directoryOutputWindowPane.Clear()
    For Each file In files
        'Display the file path, with a line break between each line
        directoryOutputWindowPane.OutputString(file + Chr(13))
    Next
End Sub
```

This macro demonstrates the use of a few methods and properties of the *OutputWindowPane* object. The *Activate* method makes sure the pane corresponding to the instance of the *OutputWindowPane* that it's being called on is the same pane displayed to the user; it simulates the selection of that pane from the drop-down list in the Output window. *OutputString* dumps a string into the pane, and *Clear* removes all text from that pane. Another property, *TextDocument*, which isn't shown in this macro, deserves special note. It returns an *EnvDTE.TextDocument* object for the pane that's read-only—you can retrieve the contents of this window, but not change it. (You can use *OutputString* only to modify the contents.) We'll discuss this object in further detail in the next chapter.

The Forms Designer Window

You visually create the user interface for your .NET Framework program within the Forms designer. By simply dragging and dropping controls from the Toolbox onto a form and setting a few properties in the Properties window, you can quickly build the user interface for your program. The Forms designer was built with .NET components and uses the *System.Windows.Forms* assembly to display and create the form. Because the *System.Windows.Forms* assembly is used, programming a form in the designer is similar to programming a form as it executes at run time.

The *IDesignerHost* Interface

A Forms designer window exposes an object model, as many of the other windows do. The object hierarchy returned from calling the *Window.Object* property of a Forms designer window is of type *System.ComponentModel.Design.IDesignerHost*. To examine and modify a form within the designer, you must find the *System.Windows.Forms.Control* object for that form. You can do this by calling the *IDesignerHost.RootComponent* property and casting the object returned into a *Forms.Control* object, as shown in this code snippet:

```
System.Windows.Forms.Control control;
System.ComponentModel.Design.IDesignerHost designerHost;
designerHost =(System.ComponentModel.Design.IDesignerHost)
    applicationObject.ActiveWindow.Object;
control = (System.Windows.Forms.Control)designerHost.RootComponent;
```

Note The *System.Windows.Forms.Form* class derives from the *System.Windows.Forms.Control* class. The code shown here demonstrates how to manipulate a user control, but you can use the same unmodified code to program a Windows Form.

Using the *System.Windows.Forms.Control* object, you can connect events to determine when the form was modified; to find and modify properties such as the dimensions of the form; and to place, modify, and remove controls from the form.

Marshaling

If you try to use the *IDesignerHost* interface from within a macro, a *System.Runtime.Remoting.RemotingException* exception is thrown. This is because user interface elements, such as the *System.Windows.Control* object, cannot be remoted across process boundaries. Remember that the Macros IDE runs in a process separate from the Visual Studio process. Because of this restriction, the designer object model can be used only within an add-in and not from a macro.

Adding Controls to a Form

Once you find the *IDesignerHost* interface for a Forms designer, you can easily add new controls to the form. To add a control, you need the *System.Type* object that describes the control. You can find this object by using the *Type.GetType* static method, which is passed the full class name of a control. For example, to add a list view control to a form, you can use code such as this:

```
System.Type type;
type = System.Type.GetType("System.Windows.Forms.ListView");
```

You can then pass this *Type* object to the *IDesignerHost.CreateComponent* method to create the control. This method has two overloads, the first of which takes two parameters. The first parameter is the *Type* object we just found, and the second parameter is the variable name of the control we want to create. This variable name must be unique among variables contained within the form's class; otherwise, a name collision will occur and an exception will be generated. The second overload of this method takes as an argument only the *Type* object; the Forms designer examines the form code to find a unique variable name to use. Both of these overloads emit the appropriate code to instantiate a control of the specified type. The following code creates a list view control with the variable name *listViewControl*:

```
System.ComponentModel.IComponent component;
component = designerHost.CreateComponent(type, "listViewControl");
```

If you were to add this code to an add-in and execute it, you wouldn't see the control appear on the form. This is because the control, while instantiated, hasn't been parented to the form and added to the form's *Controls* collection. To add the control to the form's *Controls* collection, you must set the *Parent* property of the control to the form that should contain the control. You can set the *Parent* property (or any property of a control, for that matter) by using the *System.ComponentModel.PropertyDescriptorCollection* object. This object contains a collection of properties available for a control; as values are set for the properties they contain, code is generated within the form's class that corresponds to the property you set. You can set the *Parent* property as follows:

```
System.ComponentModel.PropertyDescriptorCollection props;
System.ComponentModel.PropertyDescriptor propParent;
//Find the properties for the listViewControl control:
props = System.ComponentModel.TypeDescriptor.GetProperties(component);
//Get the Parent property
propParent = props["Parent"];
//Set the Parent property to the form:
propParent.SetValue(newControl, designerHost.RootComponent);
```

Finding Existing Controls

You now know how to create controls and place them on a form. But how do you find existing controls on a form? As mentioned earlier, the *IDesignerHost.RootComponent*

property returns an object that can be cast into a *System.Windows.Forms.Control* object. Using this object, you can call methods and properties just as you would at run time to find information about a form. For example, the following code walks the list of controls contained in a *System.Windows.Forms.Control* object:

```
System.ComponentModel.Design.IDesignerHost designer;
System.Windows.Forms.Control rootControl;

//Set the designer variable here from the Window.Object property

rootControl = (System.Windows.Forms.Control)designer.RootComponent
foreach (System.Windows.Forms.Control control in rootControl.Controls)
{
    //Retrieve desired control information
}
```

You can use the *PropertyDescriptorCollection* object to find properties of a control much as you would to set properties on the form, except you use the *PropertyDescriptor.GetValue* method:

```
System.ComponentModel.Design.IDesignerHost designer;
System.ComponentModel.PropertyDescriptor propControls;
System.ComponentModel.PropertyDescriptorCollection props;
System.ComponentModel.IComponent component;
System.Windows.Forms.Form form;
System.Drawing.Size size;
designer = (System.ComponentModel.Design.IDesignerHost) _
    applicationObject.ActiveWindow.Object;
component = designer.RootComponent;

//Get the Size property using the forms designer:
props = System.ComponentModel.TypeDescriptor.GetProperties(component);
propControls = props["Size"];
size = (System.Drawing.Size)propControls.GetValue(component);

//Get the Size property directly from the form:
form = (System.Windows.Forms.Form)component;
size = form.Size;
```

A Form Layout Sample

Microsoft Visual Basic versions 6 and earlier have a tool window called Form Layout that shows the size of a form being designed as it would appear on the desktop of your computer. Visual Studio doesn't have this feature, but you can easily create one by using the automation model of the Forms designer. You can find the source code for this sample, called FormView, among the book's sample files.

When the add-in starts, it creates a tool window that draws a virtual monitor representing your computer monitor. The screen of the virtual monitor matches the display resolution of your monitor. (If your computer uses multiple monitors, the resolution of the primary monitor is used.) After connecting to the *WindowActivated* event, it waits for a Forms

designer window to become active, and then it looks at the available controls in the form and draws the form and its controls on the virtual screen. For example, if you create a form that has the calendar and button controls on it, as shown in Figure 9-1, the Forms designer window, shown in Figure 9-2, appears. This sample also demonstrates our next topic, creating your own tool windows that are hosted by Visual Studio.

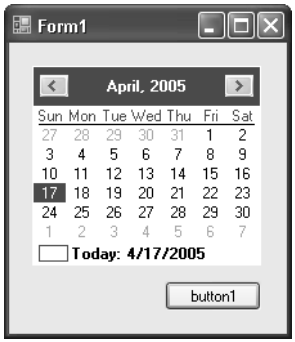


Figure 9-1 A Windows Form with a calendar control and a button control

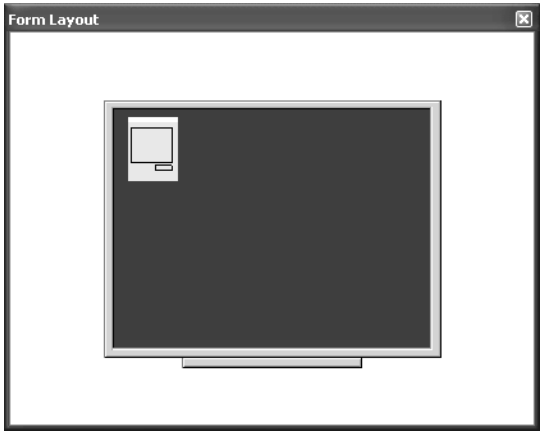


Figure 9-2 The Form Layout window showing the form from Figure 9-1

Creating Custom Tool Windows

As you know, most of the windows in Visual Studio have an object model that you can use to program the contents and present data that your code generates. However, at times you might need to display data in a way that the existing tool windows cannot handle. To allow you to display data in a way that is most suitable for your add-in, the Visual Studio object model allows creation of custom tool windows.

There are two ways to create a tool window—one way is with an ActiveX control, and the other is with a .NET User Control. To create a tool window, all you need is an ActiveX

control and an add-in that makes a call to the *Windows.CreateToolWindow* method. *CreateToolWindow* has the following method signature:

```
public EnvDTE.Window CreateToolWindow(EnvDTE.AddIn AddInInst,
    string ProgID,
    string Caption,
    string GuidPosition,
    ref object DocObj)
```

This method returns a *Window* object that behaves as any tool window that Visual Studio creates. Here are the arguments for this method:

- ***AddInInst*** An add-in object that's the sponsor of the tool window. When the sponsor add-in is unloaded, all tool windows associated with that add-in are automatically closed and the ActiveX control is unloaded.
- ***ProgID*** The ProgID of the ActiveX control that's hosted on the newly created tool window.
- ***Caption*** The text to show in the title bar of the new tool window.
- ***GuidPosition*** A GUID in string format. As you'll recall, the *Windows.Item* method can be indexed by a GUID, and that GUID uniquely identifies a specific window. The GUID assigned to your tool window and the GUID passed to the *Windows.Item* method are set by using this parameter. This GUID must be different from the GUID used by other tool windows; if you call *CreateToolWindow* multiple times, you must use a different GUID for each window.
- ***DocObject*** Most ActiveX controls have a programmable object in the form of a *COM IDispatch* interface, which is mapped to a *System.Object* when you're using the .NET Framework. The programmable object of the ActiveX control is passed back to the caller through this parameter, which allows you to program the control as you would any other tool window. You can also retrieve the programmable object of the ActiveX control by calling the *Object* property of the *Window* object for the tool window that's created by using the *CreateToolWindow* method.

To demonstrate the use of the *CreateToolWindow* method, the sample files that accompany this book include an add-in project called VSMediaPlayer. This sample creates a tool window hosting the ActiveX control for Windows Media Player and then, by using the programmable object of the control, plays an audio file. The code that does the work of creating the tool window looks like this:

```
void CreateMediaPlayerToolWindow()
{
    EnvDTE.Windows windows;
    EnvDTE.Window mediaPlayerWindow;
    object controlObject = null;
    string mediaPlayerProgID = "MediaPlayer.MediaPlayer";
    string toolWindowCaption = "Windows Media Player";
    string toolWindowGuid = "{AB5E549E-F823-44BB-8161-BE2BD5D698D8}";

    //Create and show a tool window that hosts the
    // Windows Media Player control:
```

```
    windows = applicationObject.Windows;
    mediaPlayerWindow = windows.CreateToolWindow(addInInstance,
                                                 mediaPlayerProgID,
                                                 toolWindowCaption,
                                                 toolWindowGuid,
                                                 ref controlObject);
    mediaPlayerWindow.Visible = true;

    //Play the Windows "Tada" sound:
    //Can get only the system directory (Eg: C:\windows\system32),
    // need to change this to the Windows install dir
    string mediaFile = System.Environment.GetFolderPath(
        System.Environment.SpecialFolder.System);
    mediaFile += "\\..\\media\\tada.wav";
    MediaPlayer.IMediaPlayer2 mediaPlayer =
         (MediaPlayer.IMediaPlayer2)controlObject;
    mediaPlayer.AutoStart = true;
    mediaPlayer.FileName = mediaFile;
}
```

The *CreateMediaPlayerToolWindow* method is called in two places in the sample add-in—once in the *OnConnection* method and once in the *OnStartupComplete* method. It must be called twice because of the way add-ins are loaded by Visual Studio. If an add-in is set to load on startup, when Visual Studio starts, the add-in starts loading. This loading process includes calling the *OnConnection* method. But the *OnConnection* method is called just before the Visual Studio main window is created and shown. If you call the *CreateToolWindow* method within *OnConnection* before the main window is shown, creating the tool window will fail because creating an ActiveX control requires its parent window to be visible. You can check to make sure that the main window has been created by examining the *connectMode* argument passed to the *OnConnection* method. If this is set to *ext_cm_AfterStartup*, the add-in was loaded through the Add-in Manager or by means other than the *load on startup* flag being set and Visual Studio being started. Therefore, the tool window can be shown when an add-in is loaded by using an *OnConnection* implementation such as this:

```
public void OnConnection(object application,
    Extensibility.ext_ConnectMode connectMode, object addInInst,
    ref System.Array custom)
{
    applicationObject = (_DTE)application;
    addInInstance = (AddIn)addInInst;

    //If the add-in is loaded from the Add-in Manager dialog, then
    // create and show the tool window:
    if(connectMode == Extensibility.ext_ConnectMode.ext_cm_AfterStartup)
    {
        CreateMediaPlayerToolWindow();
    }
}
```

If the *load on startup* flag is set and you want to show the tool window when an add-in is loaded, you can create the window in the *OnStartupComplete* method. This method is called

when initialization of Visual Studio is complete, which includes creating and showing the main window. It's as simple as this code snippet:

```
public void OnStartupComplete(ref System.Array custom)
{
    //If the add-in is loaded at startup, then
    // create and show the tool window:
    CreateMediaPlayerToolWindow();
}
```

To create a tool window by using a .NET user control, you will need to call the *Windows2.CreateToolWindow2* method. This method has a signature that is similar to the *Windows.CreateToolWindow* method, but rather than taking a COM ProgID to identify a control, *CreateToolWindow2* takes both a name of a .NET assembly and the full name of a class within that assembly that derives from the .NET Framework's *System.Windows.Forms.UserControl* class. The signature of the *CreateToolWindow2* method is:

```
public EnvDTE.Window CreateToolWindow2(EnvDTE.AddIn AddInInst,
    string Assembly,
    string Class,
    string Caption,
    string GuidPosition,
    ref object DocObj)
```

The arguments that differ from the *CreateToolWindow* method are:

- ***Assembly*** The name of an assembly. This can be either a full path to an assembly, such as *C:\MyControlAssembly.dll*, or a strong name for an assembly, such as *Microsoft.VisualStudio.Windows.Forms, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a, processorArchitecture=MSIL*, which is the name of the assembly implementing forms within the .NET Frameworks. The assembly name can also be a URL, meaning the prefix *file:///* and even *http://* can be used to locate the assembly.
- ***Class*** The fully qualified name of a class implementing *System.Windows.Forms.UserControl*. The namespace as well as the class name must be given using the dotted format, such as *System.Windows.Forms.PropertyGrid*, which is the name for the property grid, the same grid used within the Properties window inside of Visual Studio.

Note When using the *CreateToolWindow* or *CreateToolWindow2* method, you should not use code such as *Guid.NewGuid().ToString("B")* to pass the GUID string. This code will create a new GUID each time the method is called. Rather, you should declare a variable at class scope such as *string toolWindowGuid = "{6b74173d-c3e0-4d95-a6bc-877e660c319d}";* to create one GUID for each window type. If you ever need to find the Window object for the tool window again, you can locate that window directly with the *DTE.Windows.Item(toolWindowGuid)* method. Using the same GUID each time also will allow Visual Studio to save the position of the tool window wherever the user has placed the window, so the next time you create the tool window it can be placed back in the same position. You should use this GUID only for a tool window of the same type, and you should never use the same GUID to create multiple or different tool windows.

The Form Layout sample uses the *CreateToolWindow2* method to create its user interface. Because the user control implementing the user interface is contained within the same assembly as the add-in code, the assembly name can be found by using the *Location* property of the *System.Reflection.Assembly* class. This *Assembly* class is found with the *GetExecutingAssembly* static method, and *Location* returns the full path, with the *file:///* protocol. Lastly, because the user control to be hosted is named FormLayoutCtl, and is contained within the *FormLayout* assembly, the class name passed to *CreateToolWindow2* is *FormLayout.FormLayoutCtl*.

You can also create a programmable object for a tool window created from a .NET User Control. This will allow the users of your add-in to program your tool window, just as they program the Task List window or Toolbox. To expose a programmable object, all you need to do is to define an interface, and then implement that interface on the class that derives from *System.Windows.Forms.UserControl*. This bit of code shows an interface named *IProgrammableObject*, providing a programmable object on a control named *UserControl1*.

```
public interface IProgrammableObject
{
    void Method();
}

public partial class UserControl1 : UserControl, IProgrammableObject
{
    public UserControl1()
    {
        InitializeComponent();
    }

    public void Method()
    {
        System.Windows.Forms.MessageBox.Show("Method");
    }
}
```

With this code to implement a programmable object, you can create the tool window and call *Method* like this:

```
object obj = null;
EnvDTE80.Windows2 window2 = (EnvDTE80.Windows2)applicationObject.Windows;
string thisAssembly = System.Reflection.Assembly.GetExecutingAssembly().Location;
EnvDTE.Window win = window2.CreateToolWindow2
(addInInstance, thisAssembly, "MyAddin.UserControl1",
"Window Caption", "{0d3619e3-b0b4-4af3-9053-95a29222159b}",
ref obj);
win.Visible = true;
IProgrammableObject programmableObj = (IProgrammableObject)obj;
programmableObj.Method();
```

Setting the Tab Picture of a Custom Tool Window

When two or more tool windows are tab-linked together, an image is displayed so the user can quickly recognize the tool windows that are linked together. To set the tab

picture for a tool window that's created by an add-in, you use the *Window.SetTabPicture* method. *SetTabPicture* takes as its argument a COM *IPictureDisp* type, a bitmap handle, or a path to a bitmap file (a file with the extension .bmp) such as C:*somebitmap*.bmp. To create an *IPictureDisp* object, you can use the same technique described earlier of calling the *OleLoadPictureFile* method and then passing the returned *IPictureDisp* object to the *SetTabPicture* method. Bitmap handles can be retrieved by loading a bitmap file by using any of the various ways that an image can be loaded into an instance of the .NET Framework's *Bitmap* class (such as from a resource embedded within the add-in's assembly), then the *Bitmap.Handle* property is called to retrieve the handle.

The bitmap to place onto a tool window tab must have one of two specific formats, and any deviation from these formats can cause the bitmap to appear with incorrect colors or not appear at all. The first format is for a 16-color bitmap, and it must be 16 by 16 pixels. If any portion of the bitmap is to appear transparent, the transparent pixels must have the RGB value 0,254,0. The format for this bitmap is the same format used for displaying custom pictures on command bar buttons; a bitmap can be shared for these two uses. The other format is for high-color bitmaps; it must use 24-bit color, and it, too, needs to be 16 by 16 pixels. If any portion of this bitmap is to appear transparent, the color to use is to have the RGB value 255,0,255.

You can call the *Window.SetTabPicture* method only on a tool window created by using the *Windows.CreateToolWindow* method. Windows defined by Visual Studio already have their bitmaps set; if you try to change them, an exception will be generated. If you want to set the bitmap for your own tool window, you should set it before setting the *Visible* property of your window to *true*; otherwise, the picture might not be displayed immediately. Lastly, if a custom picture is not set, Visual Studio uses a default picture—the Visual Studio logo.

Setting the *Selection* Object

As you select different windows in Visual Studio, you see the Properties window update itself with properties available for those windows. For example, if you select a file in Solution Explorer, a set of properties is made available—such as the file path, when the file was modified, or how the file should be built. When you create a tool window, you might also want to have properties for your tool window appear in the Properties window. You set items to appear in the Properties window by using the *Window.SetSelectionContainer* method, which takes as a parameter an array of type *System.Object*. These items are displayed in the Properties window when the window that has this method called on it becomes the active window. The sample VSMediaPlayerAdv, an extension to the VSMediaPlayer sample, displays a property set in the Properties window by calling the *SetSelectionContainer* method with the programmable object of Windows Media Player, which was returned through the *DocObj* parameter of the *CreateToolWindow* method. This portion of code shows how this is done:

```
object []propertiesWindowObjects = {mediaPlayer};
mediaPlayerWindow.SetSelectionContainer(ref propertiesWindowObjects);
```

You can call the *SetSelectionContainer* method only on tool windows that you create. If you call this method on a *Window* object for, say, the Solution Explorer tool window, an exception will be generated.

The Options Dialog Box

Developers can be a finicky bunch—they want Visual Studio to work the way they want down to the finest detail; if even one option is set up in a way they didn't expect, they can become quite unproductive. The Options dialog box is full of options that you can configure—everything from how many spaces are inserted when the Tab key is pressed in the text editor to whether the status bar is shown along the bottom of the main window of Visual Studio.

Changing Existing Settings

Many settings in the Options dialog box can be controlled through the automation model by using the *Properties* and *Property* objects. To find a *Properties* collection, you must first calculate the category and subcategory of the settings you want to modify. On the left side of the dialog box is a tree view control that's rarely more than two levels deep. The top-level nodes in this tree, such as Environment, Source Control, and Text Editor, are the categories of options you can manipulate. Each category contains a group of related Options pages, and each page contains a number of controls you can manipulate to customize your programming environment. The subitem nodes are the subcategories of the Options dialog box; if you select one of these nodes, the right side of the Options dialog box changes to show the options for that category and subcategory. The category and subcategory used to find a *Properties* collection are based on the category and subcategory displayed in the Options dialog box user interface, but their names might be slightly different from the category and subcategory names. To find the list of categories and subcategories, you must use the Registry Editor. First, you find the item in the Options dialog box that you want to edit. For our example, we'll modify the tab indent size of the Visual Basic source code editor, which is found on the page of the Text Editor category and Basic subcategory.

Note The Text Editor category is a bit different from other categories in the Options dialog box in that it has three levels, with the third level being a sub-subcategory. However, in the automation model, the General and Tabs sub-subcategories are combined into one and have the same name as they do in the programming language.

After running regedit.exe, navigate to the key *HKEY_ LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\8.0\AutomationProperties*. Underneath this key is a list of all the property categories accessible to a macro or an add-in. We're looking for the Text Editor category—the key whose name most closely matches this category name in the user interface is TextEditor (without a space). After expanding this item in the Registry Editor, you'll see a

list of subcategories; one of those subcategories, Basic, matches the subcategory displayed in the user interface of the Tools Options dialog box, so this is the subcategory we'll use.

Now that we've found the automation category TextEditor and subcategory Basic, we can plug these values into the *DTE.Properties* property to retrieve the *Properties* collection:

```
Sub GetVBTextEditorProperties()
    Dim properties As Properties
    properties = DTE.Properties("TextEditor", "Basic")
End Sub
```

The last step in retrieving a *Property* object is to call the *Item* method of the *Properties* collection. The *Item* method accepts as an argument the name of the property, but this name is not stored anywhere except within the object model. Remember that the *Properties* object is a collection, and, as with all other collection objects, it can be enumerated to find the objects it contains and the names of those objects. You can use the following macro to examine the names of what will be passed to the *Properties.Item* method. The macro walks all the categories and subcategories listed in the registry and then uses the enumerator of the *Properties* collection to find the name of the *Property* object contained in that collection. Each of these category, subcategory, and property names are then inserted into a text file that the macro creates:

```
Sub WalkPropertyNames()
    Dim categoryName As String
    Dim key As Microsoft.Win32.RegistryKey
    Dim newDocument As Document
    Dim selection As TextSelection
    'Open a new document to store the information
    newDocument = DTE.ItemOperations.NewFile("General\Text File").Document
    selection = newDocument.Selection
    'Open the registry key that holds the list of categories:
    key = Microsoft.Win32.Registry.LocalMachine
    key = key.OpenSubKey( _
       "SOFTWARE\Microsoft\VisualStudio\8.0\AutomationProperties")
    'Enumerate the categories:
    For Each categoryName In key.GetSubKeyNames()
        Dim subcategoryName As String
        selection.Insert(categoryName + vbLf)
        'Enumerate the subcategories:
        For Each subcategoryName In _
            key.OpenSubKey(categoryName).GetSubKeyNames()
            Dim properties As Properties
            Dim prop As [Property]
            selection.Insert("  " + subcategoryName + vbLf)
            Try
                'Enumerate each property:
                properties = DTE.Properties(categoryName, subcategoryName)
                For Each prop In properties
                    selection.Insert("   " + prop.Name + vbLf)
                Next
            Catch
            End Try
```

```
        Next
    Next
End Sub
```

Using the output from this macro, we can find the TextEditor category and the Basic subcategory and then look in the Options dialog box for something that looks like the name Tab Size. The closest match is TabSize. Using this name, we can find the *Property* object for the Visual Basic text editor Tab Size:

```
Sub GetVBTabSizeProperty()
    Dim properties As Properties
    Dim prop As [Property]
    properties = DTE.Properties("TextEditor", "Basic")
    prop = properties.Item("TabSize")
End Sub
```

Now all that's left to do is retrieve the value of this property by using the *Property.Value* property:

```
Sub GetVBTabSize()
    Dim properties As Properties
    properties = DTE.Properties("TextEditor", "Basic")
    MsgBox(properties.Item("TabSize").Value)
End Sub
```

This macro displays the value *4*, which is the same value in the Tools Options dialog box for the Tab Size option of the Basic subcategory of the Text Editor category. You set this value the same way you retrieve the value, except the *Value* property is written to rather than read:

```
Sub SetVBTabSize()
    Dim properties As Properties
    properties = DTE.Properties("TextEditor", "Basic")
    properties.Item("TabSize").Value = 4
End Sub
```

By simply changing the category and subcategories passed to the *DTE.Properties* property and looking at the list of property names generated by the *WalkPropertyNames* macro, you can modify many of the options shown in the Tools Options dialog box.

Is It What It Says It *Is*?

When you use the Visual Studio object model, you might use the Visual Basic *Is* operator or the .NET Framework *Object.Equals* method to try to determine whether two objects are the same. But the *Is* operator and the *Equals* method might not always return what you expect because of how the Visual Studio object model was built. If you run a macro such as this:

```
Sub CompareWindowsObjects()
    Dim window1 As Window
    Dim window2 As Window
```

```
    window1 = DTE.Windows.Item(Constants.vsWindowKindTaskList)
    window2 = DTE.Windows.Item(Constants.vsWindowKindTaskList)
    MsgBox(window1 Is window2)
End Sub
```

a message box with the value *True* is displayed. When you ask for a *Window* object, the object model checks to see whether a *Window* object has been created for the specific window; if not, a new *Window* object is constructed and returned to the calling code. If a *Window* object has already been created, that object is recycled and returned to the caller. This is both a performance and memory consumption optimization because new objects are not unnecessarily created (which consumes memory) and initialized (which consumes processor time). But if you run code such as this:

```
Sub ComparePropertyObjects()
    Dim props1 As Properties
    Dim props2 As Properties
    props1 = DTE.Properties("Environment", "General")
    props2 = DTE.Properties("Environment", "General")
    MsgBox(props1 Is props2)
End Sub
```

the message box displays *False* because the *Properties* collection must be reconstructed each time you call the *DTE.Properties* property to be sure it has the most up-to-date information.

Calling the *DTE.Properties* property multiple times can cause huge memory consumption problems. Suppose you call the *DTE.Properties* property repeatedly in a tight loop; every time the property is called, a new *Properties* collection is created, initialized, and then returned to the calling code. This object consumes memory for the COM object that Visual Studio creates, and if you're using a programming language supported by the .NET Framework, a .NET wrapper class that allows you to program this object is constructed. You can see the memory consumption grow almost boundlessly if you run the following macro and watch the vsmsvr.exe process (the process that hosts the instance of the .NET Framework and runs macro code) on the Processes tab of Windows Task Manager:

```
Sub RepeatedConstruct()
   Dim i As Long
    Dim props As Properties
    For i = 1 To Long.MaxValue
        props = DTE.Properties("Environment", "General")
    Next
End Sub
```

When you run this macro, the loop never allows a garbage collection to occur because the .NET Framework is focused on running your code, not searching and removing unused objects. To make sure your program doesn't consume more memory than it should, you should be sure not to create more objects than necessary. You can do so by using the *Is* operator or the *Object.Equals* method and optimizing accordingly. For example, you can rewrite the *RepeatedConstruct* macro as follows and avoid system memory stress by

simply moving the call to *DTE.Properties* outside of the loop:

```
Sub OptimizedRepeatedConstruct()
    Dim i As Long
    Dim props As Properties
    Dim showStatusbar As Boolean
    props = DTE.Properties("Environment", "General")
    For i = 1 To Long.MaxValue
        showStatusbar = props.Item("ShowStatusBar").Value
    Next
End Sub
```

An unscientific measurement (consisting of opening Windows Task Manager and noting the amount of memory consumed before and after running the macro) shows that moving the one line outside of the loop saves almost 35 MB of memory—something your users will appreciate.

Creating Custom Settings

Not only can you examine and modify existing settings, but you can also create your own options for your add-ins. Creating a page in the Options dialog box for your add-in requires a .NET user control and creating an .addin file to let Visual Studio know how to load your Options page. The .addin file that you create for a tools options page does not necessarily need to contain the XML code to declare an add-in, but it can. When the user opens the Options dialog box, all the .addin files that can be found are opened, and, if the settings for a custom tools options page is found, the .NET user control is instantiated and shown in the Options dialog box.

Creating a tools options page is rather easy, especially with the CustomToolsOptionsPage starter kit included with the samples for this book. This starter kit will create both the .addin file and code for a user control that can be hosted on the Tools Options dialog box. Once you create this project, all you need to do is copy the .addin file and the .dll implementing the control into one of the special directories that Visual Studio looks for .addin files, and then start a new instance of Visual Studio. Let's look at the code that the starter kit will generate for you.

Declaring the XML for a Tools Options Page

The XML in the .addin file to declare a tools options page is quite simple; here is a snippet from an .addin file that declares a page:

```
<ToolsOptionsPage>
    <Category Name="My Custom Category">
        <SubCategory Name="My Custom Subcategory">
            <Assembly>MyCustomPage.dll</Assembly>
            <FullClassName>MyCustomPage.UserControl1</FullClassName>
        </SubCategory>
    </Category>
</ToolsOptionsPage>
```

This snippet of XML declares a page that will create a node in the tree on the left side of the Tools Options dialog box named My Custom Category. It will also create a node under My Custom Category named My Custom Subcategory. If the user were to select this node, the assembly MyCustomPage.dll will be loaded, and then the class *MyCustomPage.UserControl1* is instantiated and then displayed. If you used a name such as Environment as the category, this page is merged into the Environment node in the tree of the Tools Options dialog box. You can specify multiple SubCategory tags within a Category tag, and all those pages are grouped together under one top-level node.

The *IDTToolsOptionsPage* Interface

An Options page has three stages in its lifetime: creation, interaction, and dismissal. To allow your page to know about these three stages, the user control needs to implement the *EnvDTE.IDTToolsOptionsPage* interface. This interface has the following signature:

```
public interface IDTToolsOptionsPage
{
    public void GetProperties(ref object PropertiesObject);
    public void OnAfterCreated(EnvDTE.DTE DTEObject);
    public void OnCancel();
    public void OnHelp();
    public void OnOK();
}
```

When the user first displays the Options dialog box, Visual Studio sees in the .addin file that you've declared a page, and it creates an instance of your .NET user control. If the *IDTToolsOptionsPage* interface is implemented on that control, the *OnAfterCreated* method is called and is passed the *DTE* object for the instance of Visual Studio that is creating the control. The implementation of this method can perform any initialization steps needed, such as reading values from the system registry and using these values to set up the user interface of the control.

The Options dialog box has three buttons the user can click: OK, Cancel, and Help. If the user clicks OK, the *IDTToolsOptionsPage.OnOK* method is called, giving your page a chance to store back into the system registry any values that the user might have selected. You should also perform any cleanup work in the *OnOK* method because the Options page is about to be dismissed. If the user clicks the Cancel button, the *OnCancel* method is called. No values that the user selected in the page should be persisted, and this method is called so you can perform any cleanup necessary because, as when the user clicks OK, the Options dialog box is about to be closed. If the user clicks Help, the *OnHelp* method is called, giving your page a chance to display any help necessary to the user. Unlike the other buttons, Help doesn't dismiss the dialog box, so you shouldn't do any cleanup or store or discard values during this method call.

The last method of the *IDTToolsOptionsPage* interface is the *GetProperties* method. This method allows users to retrieve a *Properties* object for the options on your page in the same way they could retrieve a *Properties* object for other Options pages.

Exposing a *Property* Object

As you saw earlier, many of the values in the Options dialog box are programmable through the *Properties* collection. You can also allow the user to set and retrieve the values of your page through the *Properties* collection by using the *GetProperties* method. This method returns a *System.Object* object instance, which is wrapped up into a *Properties* collection by Visual Studio and handed back to the user when the *DTE.Properties* property is called with the category and subcategory of your page. By default, the starter kit creates one property, called *SampleProperty*, and it returns a string with the value *property value*. You can change the value, type, and name of this property, and you can also add new properties. To modify or add properties, you need to change the interface *PropertiesInterface* that was generated by the starter kit, making any changes necessary, and then reflect those properties in the class *PropertiesImplementation*, which implements the *PropertiesInterface* interface. This interface and class set-up is necessary to allow Visual Studio to properly wrap the object and return a *Properties* object to the user. You also need to use two attributes to help Visual Studio create the *Properties* object. The first of these properties, *ComVisibleAttribute*, will expose the class as a COM object. Visual Studio uses a COM type library to inspect the properties object, and this is possible only with the *ComVisibleAttribute* being set to *true*. It is not necessary to set the Register for COM interop value in the project properties; the object needs just to be visible as a COM object. The second attribute, the *ClassInterfaceAttribute* attribute, adjusts how the type library information is exposed from the class implementing the properties, and it should look like this:

```
[System.Runtime.InteropServices.ClassInterface
    (System.Runtime.InteropServices.ClassInterfaceType.AutoDual)]
```

Of course, all these attributes are set up for you by the starter kit, so you should not need to make any of these changes yourself.

To find the property exposed by this object, you will pass the category and subcategory values from the .addin file to the DTE.Properties property and then use the returned *Properties* collection just as you would for a property built-in to Visual Studio. This macro will display the *SampleProperty* property for the XML snippet given earlier:

```
Sub ShowSamplePropertyValue()
    Dim props As Properties

    props = DTE.Properties("My Custom Category", "My Custom Subcategory")
    MsgBox(props.Item("SampleProperty").Value)
End Sub
```

Looking Ahead

In this chapter, you learned how you can program many of the windows available in Visual Studio. In the next chapter, we'll show you how to program the data in one specific window type: the text editor window.

Chapter 10

Text-Editing Objects and Events

In this chapter:

Much of what a programmer does during the workday (and work night) involves editing text. In fact, editing text is so much a part of programming that many a successful business has been built around creating a better Notepad, and the popularity of these editors has grown in direct proportion to the number of mundane tasks that they automate and the extent to which they can be customized. As you learned in Chapter 3, Microsoft® Visual Studio® 2005 boasts a first-class code editor, and the automation object model lets you take advantage of the editor's functionality to create all the editing features that would have been included had you been in charge of development.

Editor Windows

If you want to edit text in the integrated development environment (IDE), you need a document; if you have a document, you also have an editor window. In Visual Studio 2005, documents and editor windows are inseparable, so it pays to know a little about editor windows even if editing text in documents is your ultimate goal. Figure 10-1 gives a sneak preview of the editor windows of interest to us in this chapter.

The *Window* Object

There's not much to say about the *Window* object—it's just a short stop on the way to more specialized windows. Finding a window is straightforward: if you want the window that has the focus, the *DTE.ActiveWindow* property returns it to you. If you want some other window and you know its caption, use *DTE.Windows.Item(<caption>)*. (Figure 10-1 shows the code for retrieving the Connect.cs and HTMLPage1.htm windows.)

After you have a *Window* object, the most important property for finding other windows is *Object*, which returns the corresponding *TextWindow* or *HTMLWindow* object for editor

windows. If you don't know for certain which type the *Object* property holds, you'll have to check by using the *TypeOf...Is* (Microsoft Visual Basic®) or *is* (C#) keyword:

```
If TypeOf DTE.ActiveWindow.Object Is TextWindow Then
    §
End If
```

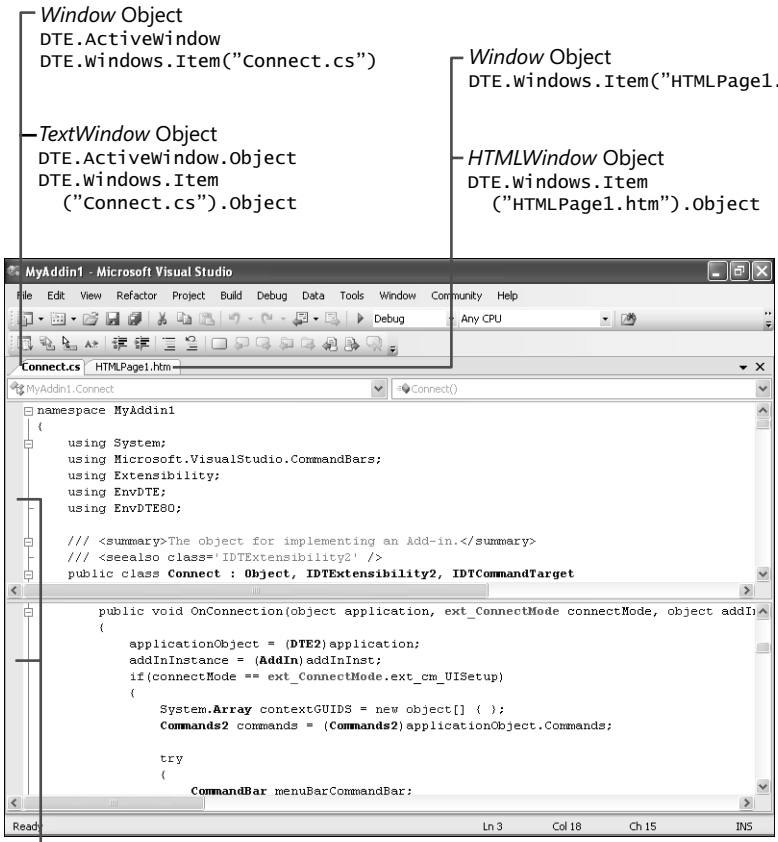


Figure 10-1 Editor windows

Of course, if you don't check and you use the wrong object, you'll receive an exception courtesy of the common language runtime (CLR).

The *TextWindow* and *HTMLWindow* Objects

The *TextWindow* and *HTMLWindow* objects represent the editor windows in the IDE. Each type offers a small set of properties that give you access to editor-window-specific features.

Table 10-1 lists the *TextWindow* properties. The two properties of note are *ActivePane* and *Panes*, which give you access to the panes in a given editor window.

Table 10-1 *TextWindow* Properties

Property	Description
ActivePane	Returns the *TextPane* object associated with the active pane.
DTE	Returns the top-level *DTE* object.
Panes	Returns a *TextPanes* collection containing the panes in the window.
Parent	Returns the parent *Window* object.
Selection	Returns the *TextSelection* object for the active pane. (It is equivalent to *Parent.Selection*.)

Essentially, an *HTMLWindow* object is just a *TextWindow* object—except when it isn't. (We'll go into more detail on that a little bit later.) Table 10-2 shows the *HTMLWindow* properties.

Table 10-2 *HTMLWindow* Properties

Property	Description
CurrentTab	Sets or returns the currently selected tab (HTML or Design)
CurrentTabObject	Returns a *TextWindow* object when the HTML tab is selected; returns an *IHTMLDocument2* interface when the Design tab is selected
DTE	Returns the top-level *DTE* object
Parent	Returns the parent *Window* object

The *CurrentTab* property uses values from the *EnvDTE.vsHTMLTabs* enumeration: *vsHTMLTabsSource* when setting or returning the HTML tab and *vsHTMLTabsDesign* when setting or returning the Design tab. The *CurrentTabObject* property returns a *TextWindow* object when the HTML tab is selected, which is why we suggested earlier that an *HTMLWindow* is just a *TextWindow* in disguise. When the Design tab is selected, however, *CurrentTabObject* returns an *mshtml.IHTMLDocument2* interface, which provides access to the Dynamic HTML (DHTML) object model of the underlying document. Be aware that the views offered by the Design tab and HTML tab aren't synchronized; changes in one view won't propagate to the other until you switch views. In practical terms, this means that you should use references only to the current view.

Note To use the *mshtml* namespace, you need its primary interop assembly: *Microsoft.mshtml.dll*. You can find this assembly at Program Files\Microsoft.NET\Primary Interop Assemblies. Add-in writers can add a reference to this assembly by browsing to it from the Add Reference dialog box; macro writers first need to copy the DLL file to the Visual Studio 2005 PublicAssemblies folder before they can access the assembly.

As you now know, it takes several steps to discover whether a text window hides inside an arbitrary window. If you think it would be nice to have a function that takes care of these steps for you, you're in luck:

```
Function GetTextWindow(ByVal win As Window) As TextWindow
    ' Description: Returns the TextWindow object for a given window,
    '              or Nothing if not a text window

    Dim txtWin As TextWindow = Nothing

    ' Check for TextWindow
    If TypeOf win.Object Is TextWindow Then
        txtWin = win.Object

    ' Otherwise, check for HTMLWindow, then TextWindow
    ElseIf TypeOf win.Object Is HTMLWindow Then
        Dim htmlWin As HTMLWindow = win.Object

        If htmlWin.CurrentTab = vsHTMLTabs.vsHTMLTabsSource Then
            txtWin = htmlWin.CurrentTabObject
        End If
    End If

    Return txtWin
End Function
```

The *TextPane* Object

The *TextPane* object represents a pane in an editor window. Every editor window can be split into two panes to allow you to juxtapose two locations in a text file. You can split the view manually either by double-clicking the splitter bar—the thin rectangle at the top of the scroll bar—or by dragging the splitter bar to the desired location. Afterward, you can make changes to the same document through either pane.

Finding *TextPane* Objects

The automation object model makes it easy to find *TextPane* objects if you already have a *TextWindow* object: just use the *ActivePane* property or iterate through the *Panes* collection until you find the *TextPane* you want. Unfortunately, the *HTMLWindow* object doesn't offer similar properties directly, so you first have to use logic like that found in the *GetTextWindow* function earlier to extract a *TextWindow* from an *HTMLWindow*.

An alternative way of retrieving a *TextPane* is through the *TextSelection* object. *TextSelection* has a *TextPane* property that returns the pane to which the selection belongs. (*TextPane* has an orthogonal property, *Selection*, that returns the *TextSelection* in the pane.) *TextWindow* and *HTMLWindow* both have a *Selection* property, as does *Window*, which means there's an indirect path to *TextPane* over which all window objects can travel. For most purposes, however, using a *TextWindow* to find a *TextPane* works just fine.

One pane-related question you might ask is whether a second pane is open in an editor window. The following code gives you the answer:

```
Function IsSecondPaneOpen(ByVal txtWin As TextWindow) As Boolean
    ' Description: Returns whether a second pane is open in a text window

    Return (txtWin.Panes.Count = 2)
End Function
```

The *TextPanes* collection returned by *Panes* has one *TextPane* object for each pane in the window, so its *Count* property returns 2 when a second pane is open.

Here's a more interesting problem—finding the top or bottom pane in a window. The problem would be intractable except for the fact that the bottom pane is always at index 1 of its *TextPanes* collection. Given that bit of information, here are two functions that return the appropriate pane:

```
Function GetTopPane(ByVal txtWin As TextWindow) As TextPane
    ' Description: Returns the top pane in the text window

    Dim txtPane As TextPane = Nothing

    If txtWin.Panes.Count = 1 Then
        ' Only one pane, so return it
        txtPane = txtWin.ActivePane
    Else
        ' Top pane is always index 2
        txtPane = txtWin.Panes.Item(2)
    End If

    Return txtPane
End Function

Function GetBottomPane(ByVal txtWin As TextWindow) As TextPane
    ' Description: Returns the bottom pane in a text window. Returns
    '              top pane if only one pane is open

    ' Bottom pane is always index 1
    Return txtWin.Panes.Item(1)
End Function
```

The *ActivateTopPane* and *ActivateBottomPane* macros included with the book's sample files let you test the previous code on live windows.

One last question you might have is which pane a given *TextPane* belongs to. At first, it might seem easy enough to compare the given *TextPane* with its corresponding member in the *TextPanes* collection, but for the reasons given in the Chapter 9 sidebar ("*Is* It What It Says It *Is*?") you can't compare *TextPane* references for equality. Fortunately, you can compare *TextSelection* references successfully, an operation that provides all the information you need to write the following functions:

```
Function IsTopPane(ByVal txtPane As TextPane) As Boolean
    ' Description: Returns whether the given TextPane is the top pane
```

```
    Dim result As Boolean = False

    If txtPane.Collection.Count = 1 Then
        result = True
    Else
        If txtPane.Selection Is txtPane.Collection.Item(2).Selection Then
            result = True
        End If
    End If

    Return result
End Function

Function IsBottomPane(ByVal txtPane As TextPane) As Boolean
    ' Description: Returns whether the given TextPane is the bottom pane

    Dim result As Boolean = False

    If txtPane.Collection.Count = 2 Then
        result = _
            (txtPane.Selection Is txtPane.Collection.Item(1).Selection)
    End If

    Return result
End Function
```

Documents

At the risk of stating the obvious (and possibly the painfully obvious), the Visual Studio 2005 text editor operates on documents. When you program, it's easy to think that you're typing in a file; you load source code from a file when you begin editing and you save the changes to a file when you finish, so it's natural to assume that all the time in between is spent working on a file. However, a file is something that exists on disk—the document you work with in the text editor is something less permanent but infinitely more malleable. This section introduces you to the two objects that capture these qualities of documents and make them available to you through automation: the *Document* and *TextDocument* objects.

The *Document* Object

The *Document* object serves as a general-purpose wrapper for text data; it provides methods and properties that give you high-level control over both the data and the windows in which that data appears.

Creating and Finding Documents

You can create a document programmatically by using methods of the *ItemOperations* object, which is covered in Chapter 7, and the *ProjectItems* object, which is covered in Chapter 8. For example, the *ItemOperations.NewFile* method, which corresponds to the New command on the File menu, lets you create a file that isn't associated with a particular project. The

following macro shows how to create a text file by using the *NewFile* method:

```
Sub CreateNewTextFile()
    ' Description: Shows how to use the ItemOperations.NewFile method
    '              to create a new text file

    Dim Item As String = "General\Text File"
    Dim Name As String = "MyTextFile"
    Dim ViewKind As String = Constants.vsViewKindPrimary
    Dim win As Window

    win = DTE.ItemOperations.NewFile(Item, Name, ViewKind)
End Sub
```

One peculiarity of the *NewFile* method's *Name* parameter is that it specifies the name of the new document indirectly. With an existing file, the document name and the window caption both correspond to the file name. With a document created by *NewFile*, however, the *Name* parameter serves as the caption of the new document's window only—the document acquires the name of the temporary file created by Visual Studio 2005 to store the new document. The "indirectly" part happens when you save the document; Visual Studio 2005 displays the *Name* value as the default name of the file to save.

You have three main ways of finding and retrieving an existing *Document* object: the *DTE.Documents* collection, the *Window.Document* property, and the *DTE.ActiveDocument* property. The *DTE.Documents* collection contains a reference to every open *Document* object. Just as with any other collection in the automation object model, you can iterate through the *Document* objects in the collection looking for the one you want, or, if you know the name of the document, you can retrieve it by using the *Documents.Item* method, like so:

```
Dim doc As Document = DTE.Documents.Item("MyFile.cs")
```

If you have a *Window* object, its *Document* property returns the associated *Document* object. Some of the tests for this chapter use the following macro to retrieve the *Document* object of a *Window*; if the window doesn't exist, the macro creates a new text file with the requested caption and returns its *Document* object:

```
Function GetDocument(ByVal caption As String) As Document
    ' Description: Retrieves the Document object associated with
    '              the specified window, or creates a text file in
    '              a new window and returns its Document object

    Dim win As Window

    Try
        win = DTE.Windows.Item(caption)
    Catch ex As System.Exception
        win = DTE.ItemOperations.NewFile("General\Text File", caption)
    End Try

    Return win.Document
End Function
```

Managing Document Windows

The relationship of *Document* objects to windows is one-to-many; a window always has one associated *Document*, but a *Document* can be open in many windows. You can open a new window on a document by using the *Document.NewWindow* method, which works in the same way as the New Window command on the Window menu. Each of the windows associated with a particular document will have as its caption the document name followed by a colon (:) and the window number, for example, Connect.cs:1, Connect.cs:2, and so on. Because the windows have the same underlying data, changes in one window will appear in all other related windows.

Warning Visual Basic files don't support *Document.NewWindow*, so they throw a "not implemented" exception when you call this method.

The ability to have multiple windows means that you won't find a *Document.Parent* property that returns the containing window. (Which window would it return?) Instead, you can find all the windows associated with a particular document by iterating through the *Document.Windows* collection, as shown by the following macro:

```
Sub ListDocumentWindows()
    ' Description: Lists all the windows associated with
    '              each open document

    Dim pane As OutputWindowPane = _
        GetOutputWindowPane("List Document Windows")

    pane.Clear()

    Dim doc As Document

    For Each doc In DTE.Documents
        Dim win As Window

        pane.OutputString(doc.Name & " windows:" & vbCrLf)

        For Each win In doc.Windows
            pane.OutputString("    " & win.Caption & vbCrLf)
        Next

        pane.OutputString(vbCrLf)
    Next
End Sub
```

You can find the active window for the *Document* object by using its *ActiveWindow* property, which returns the active window, if applicable, or the topmost window associated with the document if none of the document's windows is active.

Warning The *Document.ActiveWindow* property has a bug—it always returns the first document window, regardless of which window has the focus.

Managing Document Changes

The coarsest means available to the *Document* object for managing changes is its *ReadOnly* property, which allows you to get or set the document's read-only state. Methods that modify a document's text throw an exception if the document is read-only, so it's worth checking the *ReadOnly* property before you make text changes.

You can undo and redo changes to a document by using the *Document.Undo* and *Document.Redo* methods, respectively. These two methods offer the same functionality as their Edit menu counterparts. The *Undo* and *Redo* methods both return a Boolean value indicating whether the operation took place.

Warning You can call *Document.Undo* or *Document.Redo* on a read-only document as many times as you want to, so long as the corresponding undo or redo stack is empty; in such cases, the method returns *False*. The problem is that you can change the *Document.ReadOnly* property on the fly, which means you can have undoable (or redoable) changes in your document when you switch from read-write to read-only. If you call *Undo* or *Redo* on a nonempty undo or redo stack of a read-only document, you get an exception.

The *Document.TextSelection* property returns the *TextSelection* object associated with the active window, or the topmost window if none of the document's windows has the focus. You can use the *TextSelection* object's myriad editing methods and properties to automate just about any editing task. (You'll learn all about *TextSelection* objects in the upcoming section titled "The *TextSelection* Object.")

Saving and Closing Documents

The *Document.Save* method saves the document and optionally lets you choose the name and the location to save to. The *Save* method throws an exception if the location you specify doesn't already exist; if you give a correct location but no name, the *Save* method uses the current name of the document. (A bug in the Microsoft Visual C++® implementation causes the *Save* method to ignore any new file name that you give it.) If you want to save every open document in one call, use the *SaveAll* method of the *Documents* collection. What you gain in convenience you give up in control—you can't specify new names or locations for the files as you can with the *Save* method.

The *Document.Close* method closes a document and also lets you pass in a *vsSaveChanges* value that signals whether to save changes (*vsSaveChanges*), discard changes (*vsSaveChangesNo*), or let the user decide whether to save changes (*vsSaveChangesPrompt*, which is the default). The *Documents* collection has a corresponding *CloseAll* method that lets you close every document and also specify a *vsSaveChanges* value to apply to every document.

The *Document.Saved* property indicates whether the document has changes that haven't yet been saved—a value of *False* means that the document has unsaved changes, as indicated by an asterisk in the document window's title bar. Essentially, this property controls whether the IDE prompts you to save a document when the document is closed. You can write to this

property, but be aware that you will lose any unsaved changes if you close a document after setting its *Saved* property to *True*.

The *TextDocument* Object

Whereas a *Document* object can represent any document in the IDE, the *TextDocument* object represents text documents only. You retrieve a *TextDocument* object by using the *Document.Object* method and passing in an empty string or a value of "*TextDocument*"; the method returns *null* or *Nothing* for nontext documents.

The most important *TextDocument* properties and methods are those related to the *TextPoint*, *EditPoint*, and *TextSelection* editing objects. The *TextDocument.Selection* property returns the text document's selection and behaves the same as the *Document.Selection* property. The *StartPoint* and *EndPoint* properties return *TextPoint* objects that mark the beginning and end, respectively, of the text document buffer. The *CreateEditPoint* method returns an *EditPoint* object at the location of the *TextPoint* passed into the method; passing in *null* or *Nothing* creates an *EditPoint* at the beginning of the document.

Note It makes little sense to call *TextDocument.CreateEditPoint* with a *TextPoint* parameter because a *TextPoint* object already has its own *CreateEditPoint* method. However, passing null to *TextDocument.CreateEditPoint* is the only way to create an *EditPoint* without first creating an intermediary point object.

Point Objects

As you might guess, a point object represents a position in a text document. The automation object model gives you three point objects to choose from: *TextPoint*, *VirtualPoint*, and *EditPoint*.

The *TextPoint* Object

The *TextPoint* object embodies the fundamental attributes of a text document location; *VirtualPoint* and *EditPoint* implement the *TextPoint* interface, so all point objects have these fundamental attributes in common. The following list gives you an idea of what these attributes might be:

- **Line information** The *Line* property returns the number of the line that contains the point.
- **Offset information** The *AbsoluteCharOffset* and *LineCharOffset* properties return the number of characters between the point and the beginning of the document, and between the point and the beginning of the current line, respectively.
- **Extreme information** The *AtStartOfDocument*, *AtEndOfDocument*, *AtStartOfLine*, and *AtEndOfLine* properties allow you to determine whether the point is at the beginning or end of a document or line.

- **Relational information** The *LessThan*, *EqualTo*, and *GreaterThan* methods let you discover the relation of one point with respect to another.

The *TextPoint* object doesn't have methods that allow you to edit text directly. Instead, you either pass these point objects to editing methods or use them to create an *EditPoint* object at a particular location, which you can then use to edit text. Table 10-3 shows you the different ways to find a *TextPoint* object.

Table 10-3 How to Retrieve a *TextPoint* Object

Returned By	Applies To
StartPoint property	*TextDocument*
	TextPane
	TextRange
EndPoint property	*TextDocument*
	TextRange

The *VirtualPoint* Object

A *VirtualPoint* object represents a point in *virtual space*, which is a text editor feature that allows the insertion point to move indefinitely past the end of a line. When you type a character at a point in virtual space, the editor automatically fills in the space between the current end of the line and the new character. You can enable virtual space for all languages by opening the Tools–Options dialog box, selecting Text Editor–All Languages–General, and selecting the Enable Virtual Space check box in the Settings area. (See Figure 10-2.)

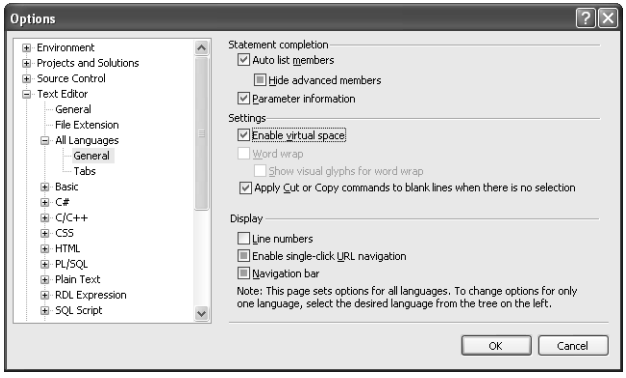


Figure 10-2 Enabling virtual space

Table 10-4 shows the different ways you can find a *VirtualPoint* object. As you can see from the table, *VirtualPoint* objects spring from *TextSelection* objects, which gives you a clue to their function; selections can extend into virtual space, so the *TextSelection* object needs *VirtualPoint* objects to keep track of endpoints that fall outside the text buffer.

Table 10-4 How to Retrieve a *VirtualPoint* Object

Returned By	Applies To
ActivePoint property	*TextSelection*
AnchorPoint property	
BottomPoint property	
TopPoint property	

As with the *TextPoint* object, one of the *VirtualPoint* object's main uses involves the creation of *EditPoint* objects. Be aware, however, that a *VirtualPoint* object can't create an *EditPoint* object in virtual space—if you try, the *EditPoint* object gets created at the end of the current line instead. You can avoid those situations by using the following function, which tells you when a *VirtualPoint* object has strayed into virtual space:

```
Function IsVirtualSpace(ByVal vrtPoint As VirtualPoint) As Boolean
    ' Description: Returns whether the VirtualPoint lies in virtual space

    Return vrtPoint.LineCharOffset <> vrtPoint.VirtualCharOffset
End Function
```

The *VirtualPoint* object defines a property named *VirtualCharOffset* that returns the distance between the point and the beginning of the line. The *VirtualCharOffset* property always has the same value as the *LineCharOffset* property, except when the point is in virtual space.

Lab: Exploring Virtual Space

The best way to understand virtual space and its effects on point objects is to test it for yourself. Here's a quick experiment:

1. Turn off virtual space in the editor.
2. Open a new text file, and type I am a fish. without pressing Enter.
3. Select the entire sentence by dragging the mouse from left to right. Notice that the editor won't extend the selection beyond the period.
4. Open the Output window and run the *DisplayTextSelectionEditPoints* macro. Observe that the *DisplayColumn* entries for *TopPoint* and *BottomPoint* are 1 and 13, respectively.
5. Run the *DisplayTextSelectionVirtualPoints* macro. Notice that the *DisplayColumn* entries for *TopPoint* and *BottomPoint* match those from the previous step.
6. Run the *DisplayTextSelectionText* macro, and observe that the output is "*I am a fish.*"

When you disable virtual space, you confine *VirtualPoint* objects to the limits of the text buffer. Enable virtual space, however, and those same *VirtualPoint* objects can wander off to parts unknown, as we can see in the following steps:

1. Enable virtual space in the text editor.

2. Reselect the entire sentence, but this time extend the selection beyond its end.
3. Rerun the DisplayTextSelectionEditPoints and DisplayTextSelectionVirtualPoints macros. Notice that the EditPoint values remain unchanged, but the VirtualPoint's VirtualCharOffset and VirtualDisplayColumn values exceed those of the corresponding LineCharOffset and DisplayColumn values.
4. Run the DisplayTextSelectionText macro, and observe that its output is the same as before.

That last step shows that virtual space exists outside the text buffer; it also shows that virtual space doesn't count as selected text. Instead, virtual space allows for what you might call WYSINWYG editing—what you see isn't necessarily what you get.

The *EditPoint* Object

The *EditPoint* is the workhorse of the point objects. In addition to the *TextPoint* methods and properties, *EditPoint* has methods that let you automate every possible modification of the text buffer. Table 10-5 shows the different ways you can retrieve an *EditPoint* object.

Table 10-5 How to Retrieve an *EditPoint* Object

Returned By	Applies To
CreateEditPoint method	*EditPoint*
	TextPoint
	VirtualPoint
	TextDocument

We'll examine the *EditPoint* object's methods shortly, in the section titled "A Comparison of the *TextSelection* and *EditPoint* objects."

The *TextSelection* Object

The *TextSelection* object pulls double duty as a representation of the caret in the editor window as well as a representation of the currently selected text. (You can think of the caret as a zero-length selection.) Because there can be only one selection in an editor window, there can be only one *TextSelection* object per document. Figure 10-3 breaks down a *TextSelection* into its constituent parts.

As you can see in Figure 10-3, four properties delineate a *TextSelection*: *TopPoint*, *BottomPoint*, *AnchorPoint*, and *ActivePoint*. Each of these properties returns a *VirtualPoint* object from one of the ends of the selected range. The *TopPoint* and *BottomPoint* properties always refer to the upper left and bottom right of the selection, respectively. The *AnchorPoint* and *ActivePoint* properties refer to the equivalent of the starting point and the endpoint of a mouse-drag selection; for example, the

top selection in Figure 10-3 would result from dragging the mouse from the beginning of *using Extensibility;* to the end of *using EnvDTE;*. You can determine the orientation of a *TextSelection* by checking its *IsActiveEndGreater* property, which returns *True* when *ActivePoint* equals *BottomPoint*. If the orientation isn't to your liking, you can flip it by calling the *TextSelection.SwapAnchor* method, which exchanges the positions of the *AnchorPoint* and *ActivePoint* objects.

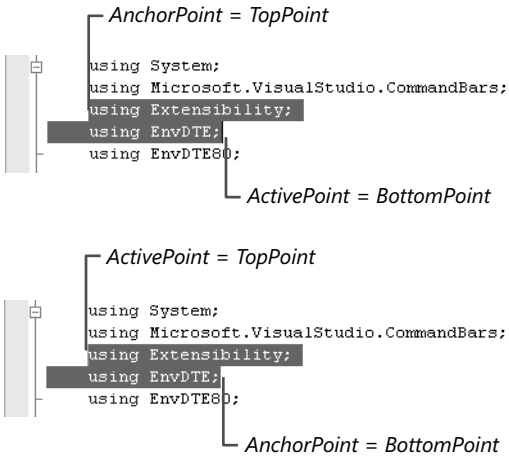


Figure 10-3 Anatomy of a *TextSelection* object

The *TextSelection.IsEmpty* property lets you know whether there's a selection, and you can retrieve the selected text from the *Text* property. If there's no selection, *Text* always returns an empty string. The converse doesn't hold, however, because *Text* returns an empty string for a virtual space selection. When a selection spans multiple lines, the *TextRanges* property holds a collection of *TextRange* objects, one for each line of the selection.

Table 10-6 lists the different ways you can retrieve a *TextSelection* object.

Table 10-6 Properties That Return a *TextSelection* Object

Property	Applies To
Selection	*Document*
	TextDocument
	TextPane and *TextPane2*
	TextWindow
	Window and *Window2*

A Comparison of the *TextSelection* and *EditPoint* Objects

The *TextSelection* and *EditPoint* objects offer a bewildering array of editing methods, which are listed in Table 10-7. Looking at the table, you'll see that *TextSelection* and *EditPoint* share the majority of their methods and have only a few seemingly minor differences, which

makes choosing one over the other akin to choosing between the 52-feature Swiss Army knife that comes with scissors and the 52-feature Swiss Army knife that comes with a saw. In most circumstances, either knife will do just fine—it's only in those particular moments when you need to gather firewood or do a little personal grooming that you suddenly realize that you can't cut down branches with scissors and you can't trim nose hairs with a saw. Using the editing objects is much the same in that you won't know whether you've chosen the right one for the job until it fails you.

Table 10-7 *TextSelection* and *EditPoint* Methods

Task	Methods in Common	*TextSelection* Only	*EditPoint* Only
Moving the insertion point	*CharLeft*, *CharRight*, *EndOfDocument*, *EndOfLine*, *LineDown*, *LineUp*, *MoveToAbsoluteOffset*, *MoveToLineAndOffset*, *MoveToPoint*, *StartOfDocument*, *StartOfLine*, *WordLeft*, *WordRight*	*Collapse*, *GoToLine*, *MoveToDisplayColumn*, *PageDown*, *PageUp*	
Finding and retrieving text	*FindPattern*	*FindText*	*GetLines*, *GetText*
Selecting text		*SelectAll*, *SelectLine*	
Modifying text	*ChangeCase*, *Copy*, *Cut*, *Delete*, *DeleteWhitespace*, *Indent*, *Insert*, *InsertFromFile*, *PadToColumn*, *Paste*, *ReplacePattern*, *SmartFormat*, *Unindent*	*DeleteLeft*, *DestructiveInsert*, *NewLine*, *Tabify*, *Untabify*	*InsertNewLine*[1], *ReplaceText*
Managing bookmarks	*ClearBookmark*, *NextBookmark*, *PreviousBookmark*, *SetBookmark*		
Miscellaneous	*OutlineSection*	*SwapAnchor*	*ReadOnly*

[1]Defined by *EditPoint2*

The fundamental difference between the two objects is that the *TextSelection* object is view-based and the *EditPoint* object is buffer-based. The *TextSelection* object exists primarily to model user actions within the text editor—if you can do it by hand in the editor, you can do it with the *TextSelection* object. (You can see this demonstrated every time you record a macro: the Macro Recorder translates changes that you make to text documents into sequences of *TextSelection* statements.) This emphasis on WYSIWYG functionality, however, means that the global view state can affect the behavior of a *TextSelection* method. For

example, when line wrapping is enabled, you can't count on *TextSelection.LineDown* to move the insertion point to the next line of text in the buffer—if the line wraps underneath the insertion point, moving the insertion point to the next line in the view serves only to move the insertion point farther down the same line in the buffer.

The *EditPoint* object, on the other hand, bypasses the view and operates on the buffer directly. Therefore, a call to *EditPoint.LineDown* always moves the *EditPoint* to the next line in the buffer regardless of the line-wrapping state. The only drawback of this insulation from the view is that you can't use *EditPoint* objects to affect virtual space.

So there you have it—if you want your add-ins and macros to make use of the view state automatically, use *TextSelection*. If you want complete control over the text buffer, use *EditPoint*.

Undo Contexts

The modern user interface has come a long way toward fulfilling one of humankind's greatest hopes—to be saved from itself. The undo facility you find in most of today's applications represents the crowning achievement of this pursuit. The next best thing to a time machine, undo allows you to roll back your most recent mistakes—usually with considerable relief—so that you can start making new ones in their place. The automation object model gives you full access to the Visual Studio 2005 undo manager, allowing you to select your own sets of mistakes that can be undone at the click of a mouse.

Automatic Undo Contexts

The basic unit of "undoability" is the *undo context*. (We'll use this term to mean both an undoable unit—the named entity that appears on the undo list—and the mechanism by which you group individual actions to create an undoable unit.) The Visual Studio 2005 IDE creates undo contexts automatically as you program, allowing you to undo and redo edits to your code. Try the following experiment to see some of the automatic undo contexts created by Visual Studio 2005:

1. Open a blank text file in Visual Studio 2005.
2. Type **spelled backwards is epyT** and press Enter.
3. Copy a block of text from some document, and paste it into the text file.

When you've finished, click the Undo button's drop-down list and you'll see the list of undo contexts shown in Figure 10-4. (The drop-down list represents the document's undo stack, which is the internal data structure that stores the undoable changes.) The three undo contexts named Paste, Enter, and Type each represent one or more individual actions that can be undone as a whole. You can appreciate the ability to group multiple actions under a single name when it comes to large paste operations, because the alternative would be undoing the pasted characters one by one.

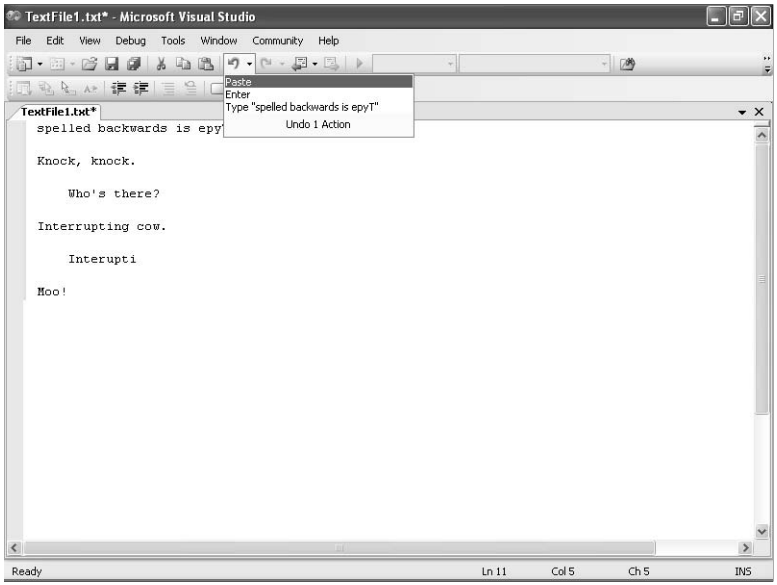


Figure 10-4 A list of undo contexts

Creating Undo Contexts

An undo context is an atomic transaction; you open the undo context and give it a name, make changes to one or more documents, and then either commit the changes by closing the undo context or abort all the changes. Once committed, the changes can be undone as a group only. You create your own undo contexts by calling methods of the *DTE.UndoContext* object: *Open* begins an undo context, *SetAborted* discards all changes made within the current undo context, and *Close* commits the changes and pushes the undo context onto the undo stacks of the participating documents.

The undo manager in Visual Studio 2005 allows only one undo context at a time to be open, and to share that undo context, you must follow a few rules. First, always call *Open* within a *try* block because this method throws an exception if an undo context is already open. Although you can check the availability of the undo context by using the *UndoContext.IsOpen* property, which returns *True* when an undo context is open, a *False* value won't guarantee that the undo context will still be free by the time your code executes *Open*. Second, if you open an undo context, you should close it when you're finished with it by calling *Close* or *SetAbort*. (Use just one or the other because *SetAbort* closes the undo context for you, and calling *Close* on a closed undo context raises an exception.) Third, you should never call *SetAbort* or *Close* on someone else's undo context.

Because only one undo context can be open at a time, if you don't acquire the undo context, any changes you make will belong to some other context. If the changes you need to make absolutely must be in their own context, you'll have to poll the *UndoContext.IsOpen* property until the undo context becomes free.

Stack Linkage

Sooner or later, when you edit multiple documents within the same undo context, you will run across the problem of desynchronized undo stacks. Suppose you edit Document1 and Document2 within the Link undo context. After you close Link, it gets pushed onto the tops of the two documents' undo stacks. Then, if you undo Link in Document1, you also undo Link in Document2 because their edits belong to the same atomic operation. So far, so good.

Suppose you add some text to Document2. These new edits get pushed onto the top of Document2's undo stack. What happens now when you try to undo Link in Document1? To respect Link's atomicity, you have to undo Link in Document2, and there's the problem—you can't undo Link in Document2 without first undoing the text that was just added. The undo stacks have become desynchronized.

The undo manager solves this synchronization problem by introducing the concept of *stack linkage*. By default, an undo context that involves more than one document has a *nonstrict stack linkage*, which allows the atomicity of the undo context to be broken across documents; when the break happens, each document ends up with its own undo context containing only changes to itself. In our previous example, if the Link undo context were created with a nonstrict stack linkage, you could undo Link in Document1 without affecting Document2. Link would disappear from Document1's undo stack but remain on Document2's undo stack, minus the changes to Document1. A *strict stack linkage*, on the other hand, enforces the undo context's atomicity. If our previous example were to involve a strict stack linkage, the undo manager would cancel any attempt to undo Link in Document1.

You specify whether the stack linkage is strict through the second parameter to *UndoContext.Open*, passing *True* for strict. You can identify undo contexts with strict stack linkage by the plus (+) sign that precedes their names on undo lists.

Lab: Strict and Nonstrict Stack Linkage

The *UndoContexts.StackLinkage* macro lets you test the differences between strict and nonstrict stack linkages. This macro creates three documents and adds text to them within an undo context; an optional Boolean parameter controls whether the undo context's stack linkage is strict. Follow these steps to see a nonstrict stack linkage in action:

1. In the Visual Studio Command window, type **Macros.InsideVS.Chapter10.UndoContexts.StackLinkage** and press Enter. The macro creates three files—Nonstrict1, Nonstrict2, and Nonstrict3—and adds text to them within the NonstrictLinkage undo context.
2. In any of the files, click the Undo button and then click the Redo button. You'll see that the changes to the documents are undone and redone as a group.
3. Add some additional text to the Nonstrict2 file.
4. Select the Nonstrict3 file, and click its Undo button.

The changes disappear from Nonstrict3 and its Undo button dims. The undo lists for Nonstrict1 and Nonstrict2 still show NonstrictLinkage, however, which means that the atomicity of NonstrictLinkage has been broken. You'll find that Nonstrict1's NonstrictLinkage undoes the changes in Nonstrict1 without affecting Nonstrict2, and vice-versa.

Now, close all the documents and redo the previous steps, but this time add **True** to the macro command in step 1. The *True* parameter tells *StackLinkage* to create files named Strict1, Strict2, and Strict3, and to add text to them within the StrictLinkage undo context. This time, when you try step 4, you'll get the error message "The application cannot undo." That's the essence of strict stack linkage.

Text Editor Events

Whereas text-editing objects tell you what you can do within the editor windows, text editor events tell you when you can do it. The automation object model defines several events that allow you to monitor editor window activities and take action based on what you find out. Together with text editor objects, text editor events allow you to achieve hands-free control over every important aspect of the editor windows.

The *BeforeKeyPress* and *AfterKeyPress* Events

The first two text editor events—*BeforeKeyPress* and *AfterKeyPress*—are new to Visual Studio 2005. Both of these events allow you to examine and act on the user's keystrokes in real time, which makes possible a vast selection of dynamic editing features.

The *BeforeKeyPress* and *AfterKeyPress* events have three parameters in common: *KeyPress*, *Selection*, and *InStatementCompletion*. The *KeyPress* parameter give you a string representation of the character that fired the event. The keypress events fire in response to any alphanumeric characters that the user types; in addition, the events fire for the Backspace, Delete, Space, Tab, and Enter keys, and Ctrl+Enter key combination.

Warning The keypress events don't fire for just keyboard input—they also fire once for each character added through the *TextSelection.Text* property. If you're tempted to use the *TextSelection.Text* property inside a keypress event handler, take care to guard against a runaway recursion.

The *Selection* parameter gives you a reference to the active document's *TextSelection* object, which you can use to determine the location of the keypress and whether the keypress replaces existing text. Finally, the *InStatementCompletion* parameter warns you when the keypress coincides with an IntelliSense® statement completion.

In addition to the aforementioned event parameters, the *BeforeKeyPress* event has a *CancelKeypress* parameter. As you might guess, setting this parameter to *True* stops the keypress from ever reaching the editor window (and, consequently, prevents the *AfterKeyPress* event from firing).

To get a feel for how these keypress events work, Listing 10-1 shows how to implement keypress event handlers as macros.

Listing 10-1 Handling the *BeforeKeyPress* and *AfterKeyPress* events

```
<System.ContextStaticAttribute()> Public WithEvents _
    TextDocumentKeyPressEvents As EnvDTE80.TextDocumentKeyPressEvents

Private DTE2 As EnvDTE80.DTE2 = CType(DTE, EnvDTE80.DTE2)
Private output As OutputWindowPane

Public Sub EnableKeyPressEventMacros()
    ' Description: Creates an Output window pane and initializes
    '              the TextDocumentKeyPressEvents variable.

    Try
        output = _
            DTE2.ToolWindows.OutputWindow.OutputWindowPanes.Item( _
            "KeyPress Events")
    Catch ex As System.Exception
        output = _
            DTE2.ToolWindows.OutputWindow.OutputWindowPanes.Add( _
            "KeyPress Events")
    Finally
        output.Activate()
    End Try

    TextDocumentKeyPressEvents = _
        DTE2.Events.GetObject("TextDocumentKeyPressEvents")
End Sub

Public Sub DisableKeyPressEventMacros()
    ' Description: Clears the Output window pane and resets the
    '              TextDocumentKeyPressEvents variable.

    TextDocumentKeyPressEvents = Nothing

    If Not IsNothing(output) Then
        output.Clear()
        output = Nothing
    End If
End Sub

Private Sub TextDocumentKeyPressEvents_BeforeKeyPress( _
    ByVal Keypress As String, _
    ByVal Selection As EnvDTE.TextSelection, _
    ByVal InStatementCompletion As Boolean, _
    ByRef CancelKeypress As Boolean) _
    Handles TextDocumentKeyPressEvents.BeforeKeyPress
```

```
    ' Description: Handles the BeforeKeyPress event by displaying
    '              the event's parameters.

    output.OutputString("BeforeKeyPress: ")
    output.OutputString("KeyPress = " & Keypress & ", ")
    output.OutputString("Selection = '" & Selection.Text & "', ")
    output.OutputString("InStatementCompletion = " & _
        InStatementCompletion.ToString)
    output.OutputString(vbCrLf)

    If KeyPress.ToUpper() = "W" Then
        CancelKeypress = True
        output.OutputString(vbCrLf)
    End If
End Sub

Private Sub TextDocumentKeyPressEvents_AfterKeyPress( _
    ByVal Keypress As String, _
    ByVal Selection As EnvDTE.TextSelection, _
    ByVal InStatementCompletion As Boolean) _
    Handles TextDocumentKeyPressEvents.AfterKeyPress
    ' Description: Handles the AfterKeyPress event by displaying
    '              the event's parameters.

    output.OutputString("AfterKeyPress: ")
    output.OutputString("KeyPress = " & Keypress & ", ")
    output.OutputString("Selection = '" & Selection.Text & "', ")
    output.OutputString("InStatementCompletion = " & _
        InStatementCompletion.ToString)
    output.OutputString(vbCrLf & vbCrLf)
End Sub
```

If you run the *EnableKeyPressEventMacros* in Listing 10-1 and start typing in an editor window, you'll see the values of the *BeforeKeyPress* and *AfterKeyPress* event parameters displayed in the Output window. To show how *CancelKeypress* might be used to make mischief, the *BeforeKeyPress* event handler effectively removes the W key without having to pry it loose from the keyboard.

The *LineChanged* Event

The automation object model defines a third event specific to editing: the *LineChanged* event. If your application needs only to process text line-by-line, sampling every keystroke might be overkill. In those instances, the *LineChanged* event offers a low-overhead alternative to the keystroke events.

The *LineChanged* event has three parameters to help you figure out why the event fired. The first two parameters, *StartPoint* and *EndPoint*, mark the beginning and end of the changes to the text buffer. You can use these *TextPoint* values to retrieve the changes, like so:

```
Dim text As String
text = StartPoint.CreateEditPoint.GetText(EndPoint)
```

The third parameter, *Hint*, is a bit flag that holds values from the *vsTextChanged* enumeration (shown in Table 10-8). The flags set in *Hint* are evidence that you can piece together to re-create the actions leading up to the event. (In practice, the *Hint* parameter doesn't give you quite enough information to figure out exactly what led to the event—but then, if it did, it wouldn't be called a hint.)

Table 10-8 The *vsTextChanged* Enumeration

Field	Description
vsTextChangedMultiLine	The changes affected multiple lines of text.
vsTextChangedSave	The changes were saved to disk.
vsTextChangedCaretMoved	The insertion point moved off the line containing the changes.
vsTextChangedReplaceAll	The entire text buffer was replaced by an insertion.
vsTextChangedNewLine	A new line was entered.
vsTextChangedFindStarting	A find operation moved the insertion point off the line containing changes.

The *LineChanged* event doesn't really fire when the line changes—that is, it doesn't fire for each new character added to or deleted from a line. Instead, the event fires when changes to a line are committed in some way, such as when the insertion point moves off the line, changes are saved to disk, or the document window loses focus. An undo context effectively disables this event until the undo context closes; afterward, the event fires if any of the changes made within the undo context would have caused it to fire under normal circumstances (the insertion point moves off a changed line, the entire text buffer is replaced by an insert, and so forth). The event handler receives *StartPoint* and *EndPoint* values that reflect all uncommitted changes from before and during the undo context.

Looking Ahead

In this book, we've presented a range of topics related to the use and customization of Visual Studio 2005. As you explore this amazing tool and the automation object model, you'll probably start to see completely new and exciting ways that you can customize and automate the IDE. We sincerely hope that we'll start seeing solutions that have in some way been helped along by the ideas and topics discussed here.

Index

B

F

G

H

I

N

O

P

Q

T

U

V

W

X-Z

About the Authors

Brian Johnson has worked at Microsoft for six years and is the Visual Studio Content Strategist for MSDN. Before coming to Microsoft, he worked as a freelance writer and served as a journalist in the U.S. Marines. Brian lives in Redmond, Washington, with his wife, Kathy, and their three lovely children, Will, Hunter, and Buffy. You can reach him at *brianjo@microsoft.com*, and you can read his blog at *http://bufferoverrun.net*.

Craig Skibo, who lives in Redmond, has been with Microsoft for nearly 10 years, has been working on the Microsoft® Visual Studio® suite of tools since version 4.1, and often gives talks at industry conferences such as PDC and TechEd.

Marc Young works as a developer in the Visual Studio User Education group. Before he became a developer, he worked as a programming editor at Microsoft Press and as a writer for the Visual Studio User Education group. Marc lives in Seattle with his beautiful wife, Julia, cute-as-a-button son, Max, and hell-on-wheels daughter, Brigitte.

Additional Resources for Web Developers

Published and Forthcoming Titles from Microsoft Press

Microsoft® Visual Web Developer™ 2005 Express Edition: Build a Web Site Now!
Jim Buyens • ISBN 0-7356-2212-4

With this lively, eye-opening, and hands-on book, all you need is a computer and the desire to learn how to create Web pages now using Visual Web Developer Express Edition! Featuring a full working edition of the software, this fun and highly visual guide walks you through a complete Web page project from set-up to launch. You'll get an introduction to the Microsoft Visual Studio® environment and learn how to put the light-weight, easy-to-use tools in Visual Web Developer Express to work right away—building your first, dynamic Web pages with Microsoft ASP.NET 2.0. You'll get expert tips, coaching, and visual examples at each step of the way, along with pointers to additional learning resources.

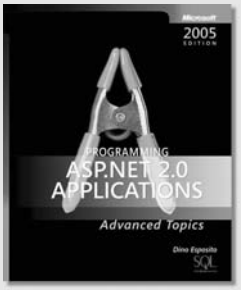
Microsoft ASP.NET 2.0 Programming *Step by Step*
George Shepherd • ISBN 0-7356-2201-9

With dramatic improvements in performance, productivity, and security features, Visual Studio 2005 and ASP.NET 2.0 deliver a simplified, high-performance, and powerful Web development experience. ASP.NET 2.0 features a new set of controls and infrastructure that simplify Web-based data access and include functionality that facilitates code reuse, visual consistency, and aesthetic appeal. Now you can teach yourself the essentials of working with ASP.NET 2.0 in the Visual Studio environment—one step at a time. With *Step by Step*, you work at your own pace through hands-on, learn-by-doing exercises. Whether you're a beginning programmer or new to this version of the technology, you'll understand the core capabilities and fundamental techniques for ASP.NET 2.0. Each chapter puts you to work, showing you how, when, and why to use specific features of the ASP.NET 2.0 rapid application development environment and guiding you as you create actual components and working applications for the Web, including advanced features such as personalization.

Programming Microsoft ASP.NET 2.0 *Core Reference*
Dino Esposito • ISBN 0-7356-2176-4

Delve into the core topics for ASP.NET 2.0 programming, mastering the essential skills and capabilities needed to build high-performance Web applications successfully. Well-known ASP.NET author Dino Esposito deftly builds your expertise with Web forms, Visual Studio, core controls, master pages, data access, data binding, state management, security services, and other must-know topics—combining definitive reference with practical, hands-on programming instruction. Packed with expert guidance and pragmatic examples, this *Core Reference* delivers the key resources that you need to develop professional-level Web programming skills.

Programming Microsoft ASP.NET 2.0 Applications: *Advanced Topics*
Dino Esposito • ISBN 0-7356-2177-2



Master advanced topics in ASP.NET 2.0 programming—gaining the essential insights and in-depth understanding that you need to build sophisticated, highly functional Web applications successfully. Topics include Web forms, Visual Studio 2005, core controls, master pages, data access, data binding, state management, and security considerations. Developers often discover that the more they use ASP.NET, the more they need to know. With expert guidance from ASP.NET authority Dino Esposito, you get the in-depth, comprehensive information that leads to full mastery of the technology.

Programming Microsoft Windows® Forms
Charles Petzold • ISBN 0-7356-2153-5

Programming Microsoft Web Forms
Douglas J. Reilly • ISBN 0-7356-2179-9

CLR via C++
Jeffrey Richter with Stanley B. Lippman
ISBN 0-7356-2248-5

Debugging, Tuning, and Testing Microsoft .NET 2.0 Applications
John Robbins • ISBN 0-7356-2202-7

CLR via C#, Second Edition
Jeffrey Richter • ISBN 0-7356-2163-2

For more information about Microsoft Press® books and other learning products, visit: **www.microsoft.com/books** *and* **www.microsoft.com/learning**

Microsoft® Press

Microsoft Press products are available worldwide wherever quality computer books are sold. For more information, contact your book or computer retailer, software reseller, or local Microsoft Sales Office, or visit our Web site at **www.microsoft.com/mspress**. To locate your nearest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in the United States. (In Canada, call **1-800-268-2222**.)

Additional Resources for Visual Basic Developers

Published and Forthcoming Titles from Microsoft Press

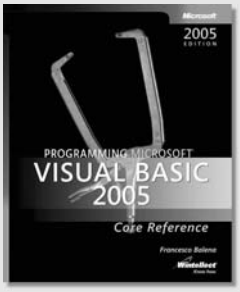
Microsoft® Visual Basic® 2005 Express Edition: Build a Program Now!
Patrice Pelland • ISBN 0-7356-2213-2

Featuring a full working edition of the software, this fun and highly visual guide walks you through a complete programming project—a desktop weather-reporting application—from start to finish. You'll get an introduction to the Microsoft Visual Studio® development environment and learn how to put the lightweight, easy-to-use tools in Visual Basic Express to work right away—creating, compiling, testing, and delivering your first ready-to-use program. You'll get expert tips, coaching, and visual examples each step of the way, along with pointers to additional learning resources.

Microsoft Visual Basic 2005 *Step by Step*
Michael Halvorson • ISBN 0-7356-2131-4

With enhancements across its visual designers, code editor, language, and debugger that help accelerate the development and deployment of robust, elegant applications across the Web, a business group, or an enterprise, Visual Basic 2005 focuses on enabling developers to rapidly build applications. Now you can teach yourself the essentials of working with Visual Studio 2005 and the new features of the Visual Basic language—one step at a time. Each chapter puts you to work, showing you how, when, and why to use specific features of Visual Basic and guiding as you create actual components and working applications for Microsoft Windows®. You'll also explore data management and Web-based development topics.

Programming Microsoft Visual Basic 2005 *Core Reference*
Francesco Balena • ISBN 0-7356-2183-7



Get the expert insights, indispensable reference, and practical instruction needed to exploit the core language features and capabilities in Visual Basic 2005. Well-known Visual Basic programming author Francesco Balena expertly guides you through the fundamentals, including modules, keywords, and inheritance, and builds your mastery of more advanced topics such as delegates, assemblies, and My Namespace. Combining in-depth reference with extensive, hands-on code examples and best-practices advice, this *Core Reference* delivers the key resources that you need to develop professional-level programming skills for smart clients and the Web.

Programming Microsoft Visual Basic 2005 Framework Reference
Francesco Balena • ISBN 0-7356-2175-6

Complementing *Programming Microsoft Visual Basic 2005 Core Reference*, this book covers a wide range of additional topics and information critical to Visual Basic developers, including Windows Forms, working with Microsoft ADO.NET 2.0 and ASP.NET 2.0, Web services, security, remoting, and much more. Packed with sample code and real-world examples, this book will help developers move from understanding to mastery.

Programming Microsoft Windows Forms
Charles Petzold • ISBN 0-7356-2153-5

Programming Microsoft Web Forms
Douglas J. Reilly • ISBN 0-7356-2179-9

Debugging, Tuning, and Testing Microsoft .NET 2.0 Applications
John Robbins • ISBN 0-7356-2202-7

Microsoft ASP.NET 2.0 *Step by Step*
George Shepherd • ISBN 0-7356-2201-9

Microsoft ADO.NET 2.0 *Step by Step*
Rebecca Riordan • ISBN 0-7356-2164-0

Programming Microsoft ASP.NET 2.0 *Core Reference*
Dino Esposito • ISBN 0-7356-2176-4

For more information about Microsoft Press® books and other learning products, visit: **www.microsoft.com/books** *and* **www.microsoft.com/learning**

Microsoft® Press

Microsoft Press products are available worldwide wherever quality computer books are sold. For more information, contact your book or computer retailer, software reseller, or local Microsoft Sales Office, or visit our Web site at **www.microsoft.com/mspress**. To locate your nearest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in the United States. (In Canada, call **1-800-268-2222**.)

Additional Resources for C# Developers

Published and Forthcoming Titles from Microsoft Press

Microsoft® Visual C#® 2005 Express Edition: Build a Program Now!

Patrice Pelland • ISBN 0-7356-2229-9

In this lively, eye-opening, and hands-on book, all you need is a computer and the desire to learn how to program with Visual C# 2005 Express Edition. Featuring a full working edition of the software, this fun and highly visual guide walks you through a complete programming project—a desktop weather-reporting application—from start to finish. You'll get an unintimidating introduction to the Microsoft Visual Studio® development environment and learn how to put the lightweight, easy-to-use tools in Visual C# Express to work right away—creating, compiling, testing, and delivering your first, ready-to-use program. You'll get expert tips, coaching, and visual examples at each step of the way, along with pointers to additional learning resources.

Microsoft Visual C# 2005 *Step by Step*

John Sharp • ISBN 0-7356-2129-2

Visual C#, a feature of Visual Studio 2005, is a modern programming language designed to deliver a productive environment for creating business frameworks and reusable object-oriented components. Now you can teach yourself essential techniques with Visual C#—and start building components and Microsoft Windows®–based applications—one step at a time. With *Step by Step*, you work at your own pace through hands-on, learn-by-doing exercises. Whether you're a beginning programmer or new to this particular language, you'll learn how, when, and why to use specific features of Visual C# 2005. Each chapter puts you to work, building your knowledge of core capabilities and guiding you as you create your first C#-based applications for Windows, data management, and the Web.

Programming Microsoft Visual C# 2005 Framework Reference

Francesco Balena • ISBN 0-7356-2182-9

Complementing *Programming Microsoft Visual C# 2005 Core Reference*, this book covers a wide range of additional topics and information critical to Visual C# developers, including Windows Forms, working with Microsoft ADO.NET 2.0 and Microsoft ASP.NET 2.0, Web services, security, remoting, and much more. Packed with sample code and real-world examples, this book will help developers move from understanding to mastery.

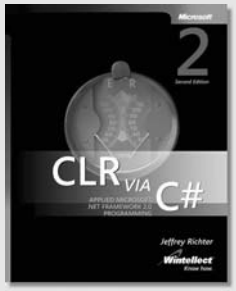
Programming Microsoft Visual C# 2005 *Core Reference*

Donis Marshall • ISBN 0-7356-2181-0

Get the in-depth reference and pragmatic, real-world insights you need to exploit the enhanced language features and core capabilities in Visual C# 2005. Programming expert Donis Marshall deftly builds your proficiency with classes, structs, and other fundamentals, and advances your expertise with more advanced topics such as debugging, threading, and memory management. Combining incisive reference with hands-on coding examples and best practices, this *Core Reference* focuses on mastering the C# skills you need to build innovative solutions for smart clients and the Web.

CLR via C#, Second Edition

Jeffrey Richter • ISBN 0-7356-2163-2



In this new edition of Jeffrey Richter's popular book, you get focused, pragmatic guidance on how to exploit the common language runtime (CLR) functionality in Microsoft .NET Framework 2.0 for applications of all types—from Web Forms, Windows Forms, and Web services to solutions for Microsoft SQL Server™, Microsoft code names "Avalon" and "Indigo," consoles, Microsoft Windows NT® Service, and more. Targeted to advanced developers and software designers, this book takes you under the covers of .NET for an in-depth understanding of its structure, functions, and operational components, demonstrating the most practical ways to apply this knowledge to your own development efforts. You'll master fundamental design tenets for .NET and get hands-on insights for creating high-performance applications more easily and efficiently. The book features extensive code examples in Visual C# 2005.

Programming Microsoft Windows Forms
Charles Petzold • ISBN 0-7356-2153-5

CLR via C++
Jeffrey Richter with Stanley B. Lippman
ISBN 0-7356-2248-5

Programming Microsoft Web Forms
Douglas J. Reilly • ISBN 0-7356-2179-9

Debugging, Tuning, and Testing Microsoft .NET 2.0 Applications
John Robbins • ISBN 0-7356-2202-7

For more information about Microsoft Press® books and other learning products, visit: **www.microsoft.com/books** *and* **www.microsoft.com/learning**

Microsoft® Press

Microsoft Press products are available worldwide wherever quality computer books are sold. For more information, contact your book or computer retailer, software reseller, or local Microsoft Sales Office, or visit our Web site at **www.microsoft.com/mspress**. To locate your nearest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in the United States. (In Canada, call **1-800-268-2222**.)