

 WILEY

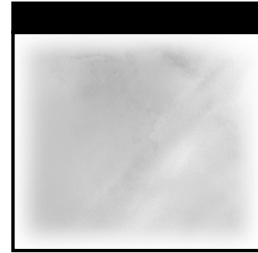
TIMELY. PRACTICAL. RELIABLE.

Developing Visual Studio® .NET Macros and Add-Ins



TEAMVIEW

Jeff Cogswell



Developing Visual Studio[®] .NET Macros and Add-Ins

Jeffrey Cogswell



WILEY

Wiley Publishing, Inc.

The
WILEY
advantage

Dear Valued Customer,

We realize you're a busy professional with deadlines to hit. Whether your goal is to learn a new technology or solve a critical problem, we want to be there to lend you a hand. Our primary objective is to provide you with the insight and knowledge you need to stay atop the highly competitive and ever-changing technology industry.

Wiley Publishing, Inc., offers books on a wide variety of technical categories, including security, data warehousing, software development tools, and networking — everything you need to reach your peak. Regardless of your level of expertise, the Wiley family of books has you covered.

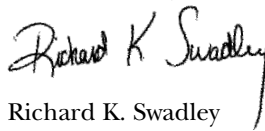
- For Dummies® – The *fun* and *easy* way™ to learn
- The Weekend Crash Course® – The *fastest* way to learn a new tool or technology
- Visual – For those who prefer to learn a new topic *visually*
- The Bible – The *100% comprehensive* tutorial and reference
- The Wiley Professional list – *Practical* and *reliable* resources for IT professionals

The book you now hold, *Developing Visual Studio®.NET Macros and Add-Ins*, is your complete guide to customizing the Visual Studio development environment. With this book, you will be able to automate routine tasks, build custom interfaces, use Office objects, and much more — using any Visual Studio-enabled language — to make this powerful development tool even more responsive and better suited to the needs of your development process.

Our commitment to you does not end at the last page of this book. We'd want to open a dialog with you to see what other solutions we can provide. Please be sure to visit us at www.wiley.com/combooks to review our complete title list and explore the other resources we offer. If you have a comment, suggestion, or any other inquiry, please locate the “contact us” link at www.wiley.com.

Thank you for your support and we look forward to hearing from you and serving your needs again in the future.

Sincerely,



Richard K. Swadley
Vice President & Executive Group Publisher
Wiley Technology Publishing

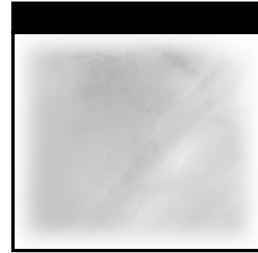


Bible

FOR
DUMMIES



Developing Visual Studio® .NET Macros and Add-Ins



Developing Visual Studio® .NET Macros and Add-Ins

Jeffrey Cogswell



WILEY

Wiley Publishing, Inc.

Publisher: Joe Wikert
Senior Editor: Ben Ryan
Developmental Editor: Adaobi Obi Tulton
Editorial Manager: Kathryn Malm
Production Editors: Micheline Frederick/Felicia Robinson
Media Development Specialist: Megan Decraene
Text Design & Composition: Wiley Composition Services

This book is printed on acid-free paper. ☺

Copyright © 2003 by Jeff Cogswell. All rights reserved.

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA, 01923, (978) 750-8400, fax (978) 646-8700. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN, 46256, (317) 572-3447, fax (317) 572-4447, e-mail: permcoordinator@wiley.com.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002. **Trademarks:** Wiley, the Wiley Publishing logo, and related trade dress are trademarks or registered trademarks of Wiley Publishing, Inc., in the United States and other countries, and may not be used without written permission. Visual Studio is a trademark or registered trademark of Microsoft Corporation. All other trademarks are the property of their respective owners. Wiley Publishing, Inc. is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data:

ISBN: 0-471-23752-3

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

*To all the .NET programmers who are
continually seeking knowledge.*

TEAMFLY



Contents

Acknowledgments	xiii
About the Author	xv
Introduction	xvii
Part 1 Automating Your Work	1
Chapter 1 All about Macros and Add-ins	3
What Are Macros and Add-ins?	4
Why Use Macros and Add-ins?	4
Automating Your Work with Macros	5
Enhancing the IDE with Add-ins	7
Online Help for Macros and Add-ins	8
Other Ways to Customize Visual Studio .NET	9
Manipulating the Windows	10
Managing the Toolbars and Commands	17
Customizing the Menus	22
Moving Forward	26
Chapter 2 Just Enough VB.NET	27
VB.NET Subroutines and Functions	29
VB.NET Variables	30
Arrays in VB.NET	34
Strings in VB.NET	36
VB.NET Control Structures	40
Conditional Statements	40
Loop Statements	41
Exceptions	43
Classes in VB.NET	44
Other VB.NET Topics	46
Moving Forward	48

Chapter 3	Introducing the Visual Studio Macro IDE	49
	The Macro System and the Main IDE	49
	Macro Development Divisions	51
	Projects and Modules	51
	Class Files and Code Files	53
	Scoping	53
	Working with the Macros IDE	54
	The Parts of the Macros IDE	55
	Managing Projects and Modules	56
	Creating a New Project	57
	Loading and Unloading Projects	57
	Saving Your Project	58
	Default Macro Location	59
	Creating a New Module	59
	Renaming Projects and Modules	59
	Deleting a Module	60
	Running a Macro	60
	Stopping a Macro	61
	Using the Code Editor in the Macros IDE	62
	Collapsible Code	62
	Working with Blocks of Code	63
	Debugging a Macro	63
	Working with the Macro Explorer	64
	Quickly Recording a Temporary Macro	66
	Selecting the Recording Project	67
	Recording a Macro	67
	Editing the Temporary Macro	68
	Assigning Shortcut Keys to Your Macros	70
	Moving Forward	71
Chapter 4	Macros That Interact with the User	73
	Referencing Assemblies and Macro Projects	73
	Referencing External Assemblies	74
	Referencing Items in Other Macro Projects	75
	Referencing One of Your Own Assemblies	77
	Creating Windows and Forms	83
	Getting Input from a User	83
	Creating a Form	84
	Displaying Standard Dialog Boxes	85
	Working with System Event Handlers	87
	Categories of Events	88
	Moving Forward	93
Chapter 5	Just Enough .NET Architecture	95
	Getting to Know Microsoft .NET	95
	Common Language Runtime	96
	.NET Class Library	97

Packaging Your Software: Assemblies and Manifests	98
Looking at an Assembly	99
Organizing Multiple Versions of an Assembly	100
.NET and OLE/COM	101
Globally Unique Identifiers	102
Building an Assembly from a COM Component	102
Macro and Add-in Models	103
Visual Studio Packages	104
Visual Studio Project Types	104
Accessing Project Types Not Supported by the Macros IDE	106
Accessing the Project Object	107
Accessing Visual Basic and C# Projects	108
Accessing C++ Projects	109
Macro IDE Automation Model	112
Moving Forward	113
Part 2 Enhancing Visual Studio	115
Chapter 6 Introducing Add-ins	117
About Add-ins	117
Startup and Shutdown	119
Invoking Your Add-in	120
Interacting with the IDE	121
Creating Custom Options Pages	123
Creating Tool Windows	124
Add-ins and the Command System	124
Wizards	125
Add-ins Are COM Components	126
Creating an Add-in	129
Using the Wizard	130
Building and Running the Add-in	136
Managing Add-ins	137
Moving Forward	138
Chapter 7 Creating Add-ins for the IDE	139
Building an Add-in in C++	139
Including a GUI with Your Add-in	140
Working with Commands	141
Using the Forms Library	141
Building a Tool Window	145
Using the Form Designer with a Tool Window	153
Moving Forward	159
Chapter 8 Life Cycles, Debugging, and Satellite DLLs	161
The Life Cycle of an Add-in	161
Debugging an Add-in	162
Keeping the Registry Clean	163
Debugging the Command-Line Version	163
Debugging the Command-Line Add-in	164
Setting Up Multiple Debug Configurations	165

	Creating a Satellite DLL for Resources	167
	About Culture-Specific Information	168
	Valid Culture Identifiers	169
	Adding Culture-Specific Resources in .NET	170
	Forms and Multiple Languages	170
	Adding Cultural Assemblies in General	171
	Making Your Add-ins Multilingual	175
	Moving Forward	184
Chapter 9	Manipulating Solutions and Projects	185
	Determining the Currently Selected Project	186
	Manipulating a Project's Items	188
	Manipulating a Project's Settings	193
	Accessing and Setting Configuration Properties	195
	Adding Configurations	198
	Configuring Projects at the Solution Level	200
	Configuring Individual Files	201
	Manipulating Language-Specific Information	205
	Moving Forward	206
Chapter 10	Programming the Document and User Interface Objects	207
	Managing Documents with the Document Classes	207
	The Form Editor and Documents	209
	Opening or Creating a Document	210
	Processing Text	212
	The EditPoint, TextPoint, and VirtualPoint Objects	218
	Working with Multiple Windows and Panes	221
	Navigating the User Interface Hierarchy	223
	Finding a Hierarchy Item	226
	Finding an Item Using Regular Expressions	226
	Selecting a Hierarchy Item	228
	Collapsing Nodes	230
	Moving Forward	231
Chapter 11	The CodeModel and Build Objects	233
	Manipulating Code with the CodeModel	234
	A FileCodeModel Add-in	237
	Working with Build Objects	248
	Spawning a Build Process	250
	Building with the SolutionBuild Object	252
	More on the SolutionBuild Object	252
	Moving Forward	254
Chapter 12	Creating Project Wizards	255
	Dissecting the Wizard Directory Structure	256
	Wizard File Interactions and Symbols	259
	Rendering the Template Files	263
	Wizard Properties	267
	A Script Wizard Tutorial	269

The common.js File	276
Viewing Your HTML Files	279
Moving Forward	281
Part 3 VS.NET and Other Products	283
Chapter 13 Writing .NET Add-ins for Microsoft Office	285
Introducing Office Add-ins	285
Writing Add-ins for Other Products	287
Preparing the Office Application	287
Creating the Add-in	288
Adding References to Office Products	291
Writing the Add-in	295
Debugging for Multiple Products	301
Responding to Office Events	303
Moving Forward	311
Chapter 14 Integrating Visual Studio .NET with Microsoft Office	313
Adding a Spelling Checker	314
Integrating an Excel Spreadsheet	322
Automating from Macros	326
Moving Forward	333
Chapter 15 Integrating with Other Products	335
Windows Script Host	335
Delphi	338
Python	340
Script Explorer Add-in	343
Moving Forward	352
Part 4 Deploying and Supercharging	353
Chapter 16 Deploying Your Macros and Add-ins	355
All about Security and Add-ins	355
.NET Security	356
Valid and Verified .NET Code	357
Security Permissions	357
Security Administration for .NET	358
Security in Action	359
COM Security	362
Deploying Macros	364
Setting Up the Deployment Project	365
Adding a Shortcut to the .vsmacros File	370
Creating the Custom Action Project	371
Moving Forward	375
Chapter 17 Supercharging Visual Studio .NET	377
Creating an Options Page for Your Add-in	377
Another Useful Add-in	388
Third-Party Add-ins	392

Exploring the CLI	393
Understanding the CLI	394
Other Languages and the CLI	394
Wrap-up	395
Appendix A Class and Object Quick Reference	397
Root DTE Object	397
DTE Properties	397
DTE Methods	400
Other EnvDTE Objects	401
Enumerations	402
Index	409



Acknowledgments

When I first set out to write this book, I seriously doubted whether the topic of macros and add-ins would be broad enough to fill some 456 pages. But as I began to develop the outline, it became clear that the information was certainly dense enough. And as I wrote page after page, I was astounded at how much there was to say about the topic. In the end, I can say honestly that this book was a blast to write.

And since writing such a book is never a one-person deal, I want to acknowledge the many people who worked hard and stayed up late to help me get this book done on time. These include Ben Ryan (acquisitions editor), who was very kind throughout this process; Adaobi Obi Tulton (development editor), who is very talented and patient; Kel Good (technical editor), who is extremely knowledgeable on all the topics covered in the book. And, of course, thanks to all the production people and additional editorial staff, including Micheline Frederick and Janice Borzendowski for the wonderful job with the final edits.

In addition to the editorial staff and John Wiley & Sons, Inc., I want to thank my dear friends Jenniffer Lesh, Jennifer and Greg Wood, and Andrea Vaduva M.D. (okay, future M.D., anyway); and my friend and agent Margot Maley Hutchison of Waterside Productions in San Diego, who, as I type this, is right now in the hospital giving birth to her second child.



About the Author

Jeff Cogswell is a software engineer by trade, and a writer and teacher at heart. He has written numerous books and articles on programming, and has taught oodles of people how to program; he enjoys seeing the light turn on for each student. He currently lives in Southern California, but as you read this he might be sitting with his laptop in a coffeehouse far from his home, possibly near your home. If you see him, be sure to say hello. Or just email him at readers@jeffcogswell.com. Also be sure to visit his Web site, www.jeffcogswell.com, where you can find even more information on .NET programming.

TEAMFLY



Introduction

Welcome to Macros and Add-ins

This is a book about developing macros and add-ins for Visual Studio .NET. Every programmer who uses Visual Studio .NET, regardless of the type of project he or she is developing, and regardless of his or her skill level, sooner or later wonders: Why can't Visual Studio .NET do such-and-such? Or: Wouldn't it be nice if Visual Studio .NET had this feature I'm thinking of? Or: There must be a way to simplify this task that I do over and over!

Surely you've thought of ways you could make Visual Studio .NET better, whether it's simply by automating a repetitive task that you have to do 158 times today or by implementing a full-scale improvement Visual Studio .NET that, for example, would add an entire window that contains suggestions for misspelled words in the comments in your code.

For the first case, simplifying repetitive tasks, you can either use a macro, which is a simple, interpreted piece of code, or an add-in, which is a compiled library that Visual Studio .NET loads and runs. In the second case, creating a new window for the Visual Studio .NET environment, you use an add-in. For numerous other tasks, you can use either a macro or an add-in.

This is all possible thanks to the developers at Microsoft, who, when they created Visual Studio .NET, included an *automation library* that provides a set of the objects and classes for manipulating the Visual Studio .NET environment. This library is available to you whether you're writing a macro or an add-in.

For the macros, Microsoft has built on the previous generation of macro tools for its products. Whereas these earlier tools featured Visual Basic for Applications (VBA), Visual Studio .NET uses Visual Basic .NET for its macro language. This means that not only can your macros drive Visual Studio .NET, but they can access the full, rich .NET framework library.

For the add-ins, Microsoft used a process that already existed in the Microsoft Office products. (For those of you who are already familiar with add-ins, I'm talking about

the IDTExtensibility2 interface that Microsoft has used in the past.) This means that if you already know how to develop add-ins for Microsoft Office, you've got a head start on developing add-ins for Visual Studio .NET. Likewise, after you read the chapters in this book on how to develop add-ins for Visual Studio .NET, you will know how to design add-ins for Microsoft Office. For this reason, I have devoted a chapter to using Visual Studio .NET to develop add-ins for Microsoft Office products.

All that said, this book does much more than just explain how to develop and use macros and add-ins. By working through the entire book, you will also learn about .NET programming in general. Specifically, you learn:

- How the Common Language Runtime works.
- What Microsoft Intermediate Language is.
- How to develop satellite DLLs that provide for globalization of your software.
- What COM is, how .NET differs from COM, and how you can make the two work together.
- And much more!

In short, at the conclusion of this book, you will have a very strong knowledge of .NET programming in general.

Required Tools

The tools you need to get the most from this book depend on what you want to do:

- *If you plan to develop add-ins*, you must have the Professional Edition, or either of the Enterprise editions of Visual Studio .NET. (Theoretically, you could develop add-ins using the Standard edition of C++, VB.NET, or C#, but you would have an uphill battle. For that reason, if you're seriously interested in developing add-ins, I recommend that you purchase a Professional or Enterprise Edition.)
- *If you plan to develop macros*, you can use any edition of Visual Studio .NET, provided it's a .NET edition. Though you can develop macros for earlier editions of Visual Studio, to take advantage of all the features I describe in this book, you will want to have a .NET edition.

Minimal system requirements include the following: You must have Windows NT 4.0 or later (that is, anything except Windows 3.1, 95, and 98; and if you're going with XP, you'll want the Professional, not the Home Edition). Microsoft also lists some requirements on its Web site; however, I recommend that you go way beyond these requirements, at least in RAM. Remember, these are *minimal* requirements, meaning Visual Studio .NET will run, but not necessarily very quickly, giving you plenty of time to go refill the coffee cup as the virtual memory loads and unloads. Here are the minimum RAM requirements Microsoft suggests:

Microsoft Windows XP Professional. 160 megabytes (MB) of RAM

Windows 2000 Professional. 96 MB of RAM

Windows 2000 Server. 192 MB of RAM

Windows NT 4.0 Workstation. 64 MB of RAM

Windows NT 4.0 Server. 160 MB of RAM

I strongly recommend that you double or even triple these numbers, especially if you're developing add-ins, because often you'll run two instances of Visual Studio .NET on a single computer. For the development of this book, I used Windows 2000 Professional with 320 Meg of RAM, and there were times my system had trouble keeping up. However, in general 320 Meg on a Win2000 box did fine. Remember, these days, RAM is pretty cheap, and if you're a serious developer, RAM is definitely something you will want to invest in. Go for 512 Meg if you can.

As for other tools, consider that programming is changing. These days, thanks to the miracle of hypertext, we have access to endless bits and bytes of documentation, all online, which we can access without having to worry about our computers grinding to a halt. And for the purposes of this book, you will need access to the online help that ships with Visual Studio .NET. So if you didn't install it already, do so before you begin reading this book, because rather than rewriting the online help, listing every single method of every single object, I provide you with an augmentation to the online help. Also, at times I refer you to the online help for more information. (But note: I do *not* assume prior knowledge of macros and add-ins, and I certainly don't expect you to read the online help before reading this book. The online help is simply a reference; this book will teach you what you need to know.)

Finally, I'll state the obvious by adding that you should have easy access to the Internet, preferably high-speed, for several reasons. First, this book has an accompanying Web site, where you can find additional information from me, more add-ins and macros, and a forum where you can share ideas and thoughts with other programmers. Second, even with the online help loaded, you will be spending a good deal of time at Microsoft's development site, <http://msdn.microsoft.com>, specifically in the library at <http://msdn.microsoft.com/library/>. If you aren't familiar with these two sites, I encourage you to take a peek at them the next time you're online, for it is at these sites that you can find nearly all the answers to anything you might ever get stuck on, well beyond the topics of macros and add-ins.

Background Knowledge

I assume you are familiar with .NET Programming basics, such as how to create solutions and projects, how to use some of the basic framework classes such as those in the IO namespace, how to run console applications, how to use the form designers, how to run the debugger, and similar information.

In addition, it will help if you know C++ programming, although most of the work in this book is in C# and VB.NET. A bit of COM and ActiveX familiarity will help, too, although for the uninitiated, I explain most of what you'll need to know of COM and ActiveX for developing add-ins. (The reason you need to know a bit about these technologies is that Visual Studio .NET is a COM automation server, and you'll be developing add-ins as COM in-process servers.)

Some knowledge of Visual Basic is also recommended, either the older versions of the language or the newer VB.NET language, because you'll be writing macros in

VB.NET. But for VB, too, I walk the uninitiated through the essentials. And I devote an entire chapter, Chapter 2, “Just Enough VB.NET,” to introducing VB.NET. And rest assured, if you’re an expert in C++ or C#, you probably will pick up VB.NET in no time, as the syntax is simple and the language is easy to learn.

I also suggest you learn a bit of C#, if you don’t know it already, because many of the examples in this book are in C#. More important, the C# language is, in all likelihood, the future of .NET. Microsoft created C# from the ground floor with .NET in mind (while borrowing a good bit of the syntax from Java and, to an extent, C++). Basically C# is the language of .NET, and so, if you’re going to develop for .NET, you will want to at least explore this exciting new language. If you know another programming language (especially C++ or Java), I guarantee you’ll be able to pick up the basics of C# in a single day. The syntax is straightforward (it’s based on C++ and Java) and the language is easy to learn.

About the Book

This book is divided into four parts:

Part 1: Automating Your Work. In Part I, I focus primarily on macro development, showing you how to automate processes in Visual Studio .NET. Even if your primary interest is add-ins, I encourage you to read this part, as there is a great deal of information overlap between macros and add-ins; most of what you can do in macros you can also do in add-ins. Moreover, in the chapters on add-ins, I assume you’ve at least given the chapters in Part I a quick read. And note that Part I includes a chapter that introduces the Visual Basic .NET language.

Part 2: Enhancing Visual Studio. Here I introduce the concepts and technology behind add-ins and then take you through the whole world of add-in development.

Part 3: VS.NET and Other Products. In this part I describe how you can use Visual Studio .NET to write add-ins for Microsoft Office, and how you can integrate Microsoft Office products such as Word, Excel, and Outlook directly into Visual Studio .NET. There are two ways of integrating: by automating the office products, using their functionality (such as checking the spelling of source comments or emailing a source file to another developer); and by embedding a spreadsheet or other Office document right into Visual Studio .NET. I wrap up Part 3 with a discussion of other tools useful in automation, such as the Windows Scripting Host and other languages such as Delphi and Python. (Yes, you can write automation macros using nearly any language you want, provided you’re comfortable stepping away from the main macro development tools built into Visual Studio .NET.)

Part 4: Deploying and Supercharging. Once you’ve developed a macro, an add-in for Visual Studio .NET, or an add-in for Microsoft Office, you’ll want to know the ins and outs of getting your product onto another computer. Part IV begins by explaining how to get your add-in deployed, and ends by describing how to supercharge Visual Studio .NET.

About the Web Site

In addition to reading this book, I invite you to visit our official Web site at www.wiley.com/compbooks/cogswell, where you'll find :

- Code examples submitted by other readers
- Pages describing additional issues you may encounter, submitted by readers such as yourself
- A discussion forum where you can talk about any issues relating to macros, add-ins, and .NET development in general
- Links to more add-ins and macros that you can download
- And much more.

The Web site was designed to help you, the software developer, get the most possible out of Visual Studio .NET as you develop add-ins and macros. Please join us!

PART

One

Automating Your Work

All about Macros and Add-ins

Macros and add-ins are nothing new. For a long time, programmers have been creating products that include features allowing users to somehow add their own customizations and enhancements to the product. Microsoft products in particular have been rich with such features for some time. For example, early versions of Word and Excel had their own macro languages, which Microsoft later replaced with Visual Basic for Applications (VBA). This included a full-featured, integrated development environment (IDE) that allowed you to design complete programs written in the VBA language. This tool was very similar to the actual Visual Basic development tool, as it enabled you to write functions in the Visual Basic language and even design forms with controls such as buttons on them. But the “for applications” part of “Visual Basic for Applications” meant that the language also made it possible to access features that were part of the program. For example, in Word, you could select text and change the font programmatically from your VBA program. In Excel, you could write data to cells and perform Excel functions such as summations.

Visual Studio .NET now has the richest macro and add-in capabilities of any previous Microsoft tool, as it uses Visual Basic .NET as its macro language. This means that in addition to the previous VBA features, you also have full access to the entire .NET foundation library.

What Are Macros and Add-ins?

Macros are small programs that automate repeated tasks. For example, in Visual Studio .NET you might write a macro that automatically saves a backup copy of your program prior to compiling for version control purposes; or you might write a macro that automatically reformats your code by reworking all the indentations and general layout, and then replaces all tabs with spaces. You might then attach these two macros to hotkeys (which Visual Studio .NET calls *keyboard shortcuts*) so that you can easily access them simply by pressing a few keys.

To write your macros, you can use a simple recorder feature. Most of the Microsoft products (including Visual Studio .NET) allow you to easily record a sequence of keystrokes and menu commands to a macro, and then play it back later. But if you want to write more sophisticated macros, you can open up the Visual Studios Macro IDE and develop macros using a full-featured IDE that looks much like the standard Visual Studio .NET IDE.

Add-ins are larger programs that do more than simply automate work. They are compiled programs that are added directly into the Visual Studio .NET process. Although you can automate work with add-ins just as you can with macros, you can write much more complex programs. In this sense, add-ins are more like enhancements to the Visual Studio .NET IDE, rather than simply an automation of a common task.

Why Use Macros and Add-ins?

Since the real purpose of Visual Studio .NET is to create software, it might seem a little strange that you can use Visual Studio .NET to create parts to use directly in that environment. The reason that feature is included is that we programmers love automation. We love to simplify our work. For example, why go through the headache of repeating the same task over and over when you can get Visual Studio .NET to do it for you? That's where the macros come in. You can easily automate common or complex tasks, and make them work for you.



You can use many of the macro techniques described in this book in other Microsoft programs, such as Microsoft Word and Microsoft Excel. They all use a form of Visual Basic for the macros, and the techniques are similar in all these products.

But there is much more beyond macros. If you want to add new features to the IDE, you can use add-ins. You can add new dialog boxes to support your new features, and you can add property sheets to the Options dialog box.

Automating Your Work with Macros

Think of macros as a way to automate Visual Studio .NET. With Visual Studio .NET's built-in macro language (which is really Visual Basic .NET, or VB.NET for short) you can write programs that manipulate the Visual Studio .NET IDE. Because the macros use VB.NET, which is a full-featured language, you have great flexibility and power in your programs.



If you're a die-hard C++ fan, as many of us are, please don't be put off by the fact that macros use VB.NET. The new VB.NET language is much more powerful than previous versions of Visual Basic. It has to be, because when you use Visual Studio .NET to create standalone VB.NET programs, Visual Studio .NET compiles it to the same intermediate language to which it compiles managed C++ program, called the Microsoft Intermediate Language, or MSIL.

But beyond the fact that you're using a full-featured language that includes all the old standards like control structures and classes, your macros also have access to the full automation engine that controls the Visual Studio .NET product. This automation engine is an *OLE Automation library* that provides control over the IDE. The object you use to access the IDE is called Development Tools Extensibility (DTE). Through this object, you have access to the full automation system for Visual Studio .NET. You will be able to access items in the Visual Studio .NET IDE such as:

- Currently open documents or just the current active document (usually a source code file).
- A list of the add-ins.
- Menu commands and available macro commands.
- Windows such as the toolbars or the tool windows. Once you have a reference to a window, you can manipulate the window.
- The solution and its projects.



Microsoft uses the term *VSMacro* to refer to a macro written for Visual Studio .NET. In this book, I generally use the term *macro*, as it's more convenient; but if I need to distinguish it from other types of macros, I use the term *VSMacro*.

Visual Studio .NET includes an entirely separate IDE for developing macros. If you've worked with the VBA system in Microsoft Word or Excel, for example, you probably have seen how this works: When you edit a macro, a new window opens that

looks very similar to the Visual Studio .NET IDE. But this window is for the development of macros. By itself, it's an entire integrated development environment for building and editing macros.



To get to the Macros IDE from the main IDE, choose Tools⇨Macros⇨Macros IDE. (Or simply press Alt-F11.) The Macros IDE will open as an entirely separate window from the main IDE.

The Macros IDE contains many of the same features you find in the usual Visual Studio .NET IDE; it also has the same look and feel. In addition, it has its own explorer, which is similar to the Solution Explorer called Project Explorer, which shows the currently loaded macro projects. But, unlike the Solution Explorer, the Project Explorer does not contain a solution; instead, it simply contains a single root node called Macros. Also, just as in the main IDE, in the Macros IDE you can access the integrated online help in the same way as the standard IDE.

Macros “live” in files with a .vsmacros extension. These files contain the VB.NET code for the macros. A single .vsmacros file can hold multiple macros; each macro lives as a subroutine.



When you set out to write a macro, you might be tempted to try writing in C++ as a standalone program. But it won't work. The reason is that the C++ program will not have access to the `DTE` object for the particular Visual Studio .NET instance you are running.

Here's a brief rundown of how to get up and running with macros. In Chapters 2 and 3, I give more detailed descriptions.

1. Open the Macros Explorer by choosing Tools⇨Macros⇨Macro Explorer (or press Alt-F8).
2. Create a new macro project. To do this, in the Macros Explorer, right-click the word “Macros” at the top of the tree and choose New Macros Project. In the dialog box that appears, choose a name and location for your macro project. (I'm calling mine “Chapter1”, since they go with Chapter 1 of this book.)
3. Double-click the name Module1. This will open the module in the Macros IDE. Here's the code that you will see:

```
Imports EnvDTE
Imports System.Diagnostics

Public Module Module1

End Module
```

There's not much there, but it's the basic starting point. It imports the `EnvDTE` namespace, which gives you access to the DTE object. It also imports the `System.Diagnostics` namespace, which gives you access to system processes and debugging information. Next are the two lines denoting the start and end of the local module, presently called `Module1`. Inside here is where you'll write your macros. For example, here's a short macro that does nothing more than minimize the Visual Studio .NET IDE window.

```
Imports EnvDTE
Imports System.Diagnostics

Public Module Module1
    Sub MinimizeMainWindow()
        Dim win As Window
        win = DTE.MainWindow()
        win.WindowState = vsWindowState.vsWindowStateMinimize
    End Sub
End Module
```

The `MinimizeMainWindow` subroutine contains the macro. This code creates a variable called `win` of type `Window`. Next the code asks the DTE object for the main window object, saving it in the `win` variable. Finally, it sets the `WindowState` member to `vsWindowStateMinimize`, which is a member of the enumerated type `vsWindowState`.

There are many ways to run the macro, but one easy way is to return to the main IDE window. In the Macros Explorer you'll see that your macro is now listed in the tree under the name `Module1`. If you double-click the name, the macro will run. (Or you can right-click and in the popup menu choose Run).

If you want to add more macros, you add more subroutines inside the module.

Enhancing the IDE with Add-ins

An add-in is a compiled program that exists as an OLE COM object. In order for Visual Studio .NET to recognize add-ins, they must implement the `IDTExtensibility2` interface.



In this book, I'm assuming that you're somewhat familiar with COM objects. In short, a COM object is one that usually lives in a DLL that implements an expected set of functions called the *interface*. In the case of add-ins, they must include the functions specified in the `IDTExtensibility2` interface. Fortunately, there's a wizard that automatically sets up the classes for you, including the function headers for the interface, so all you have to do is fill in the appropriate code. I show you how to do this starting with Chapter 6, "Creating Add-ins for the IDE."

Visual Studio .NET contains a project wizard called Visual Studio .NET Add-in Project. This wizard simplifies the process of creating an add-in by providing you with a starting point. Further, Visual Studio .NET gives you an interesting way of developing

add-ins. When you run an add-in project opened in Visual Studio .NET, a new Visual Studio .NET process will start up. Your add-in will run in this new process, which will then allow you to debug it from within the original Visual Studio .NET process.



To access the wizards that simplify the task of writing add-ins, you need to have the Visual Studio .NET Professional Edition or better (which contains C++ .NET). If you purchase the product called Visual C++ .NET, you can still write add-ins, but you do not have the wizards to help you with the task. In this book, I'm assuming that you have the Professional Edition and, therefore, have access to the Add-in wizard.

Like macros, add-ins have access to the Visual Studio .NET automation object, the DTE. This means you can completely control the Visual Studio .NET environment from within your add-in.

You have great flexibility in what you can do with add-ins. Here's a short list of some of the possibilities.

- Add new tool windows. By *tool windows*, I don't just mean a window that has features like a toolbar. Rather, all the auxiliary windows in Visual Studio .NET are tool windows. They sport tabs at their bottom, and you can drag them around so they share a window with other tool windows; and you can drop them in the middle of the IDE so they become floating windows. Examples of tool windows are the Solution Explorer, the Help index, and the Output window.
- Add new pages to the Options dialog box.
- Add new menu items and enable or disable them.

When developing add-ins, remember that you typically write them in C++, and then compile them. (However, you can also develop them in C# or VB.NET, and then compile them.) Therefore, one of the benefits of add-ins is that you can distribute them without distributing the source code. That's not the case with macros, which are interpreted from a form of VB.NET. (I go into the details of creating an add-in starting in Chapter 6, "Creating Add-ins for the IDE.")

Online Help for Macros and Add-ins

One of the most important references you will want to use when developing your macros and add-ins is the online help that accompanies Visual Studio .NET. There are dozens of objects and classes available to you for manipulating the Visual Studio .NET IDE, and these classes are all fully documented in the online help. In this book I show you how to use the majority of these classes, and I include a complete reference guide that can help you as well. But for quick answers, the online help is always your best bet.

You can open the online help in several places. First, you can open it in the IDE itself by choosing Help→Contents, Help→Index, or Help→Search. For most development work, Help→Index is typically the most useful. (There is also a dynamic help feature,

which most programmers I know don't use much, but you might find it helpful. Dynamic help tracks what you are doing and offers suggested help pages to assist you.)

You can also open the online help in its own IDE-like window. You can find this from the Start Menu, in the menu group where you installed Visual Studio .NET. The one you want is the menu item called Microsoft Visual Studio .NET Documentation. When you open this item, you will see a standalone program open that looks remarkably similar to the main Visual Studio .NET IDE. It has the same general look and feel, with a similar layout and even similar menu items; to get to the help entries, choose Help→Index or one of the other items under the Help menu. But note that this program's purpose is strictly to display the online help, not to develop programs. I like to use this program when I'm looking up topics, but for whatever reason don't have the IDE open (such as when I'm writing a book or article or doing general research). I also find it to be useful when developing macros. The Macros IDE is a separate window from the main IDE window, which enables me to put my online help in its own window as well. Then, whether I'm working in the main IDE or the Macros IDE, I just press Alt-Tab once or twice to get to the help. An added advantage is that it takes up no extra space in the IDEs. But, really, this is all a matter of taste.

Finally, you have access to the same help from within the Macros IDE, again from the Help→Index, Help→Contents, and Help→Search menus.

When searching for help on a particular class, the best way to find the help item is by opening the help index and typing the name of the class, followed by a space, and then the word Class. This will get you closest to the item. For instance, if you want help on the .NET String class and you type just String, you will find several entries for string. But if you type String Class, you will see a couple lines devoted to the ATL string class; and right after that, you'll see the string class pertaining to .NET.



When looking for information on the classes that deal with macros, sometimes typing the class name followed by the word Class doesn't quite get you to where you need to be. Instead, for most of the classes that pertain strictly to the automation engine of .NET, the entries are listed with the word Object instead of Class. Thus, the entry for DTE is listed under DTE Object, not DTE Class.

Also, when looking up information that pertains to macros, I find that the index is a bit less crowded when I open the Filtered By combobox in the help index, and choose Visual Studio Macros. Of course, the same information is available when you don't filter, but I find it a bit easier on the nerves to navigate when the filter is on.

Other Ways to Customize Visual Studio .NET

Visual Studio .NET includes many ways you can enhance the tool beyond writing macros and add-ins. For example, you can easily add items to the Tools menu. And, of course, you can arrange the windows the way you want them, and save them. Or you can easily customize the toolbars that exist in some of the tool windows.

In this book I focus primarily on macros and add-ins. However, to make your development as productive as possible, you should know how to completely manipulate the IDE in these alternate manners. Therefore, in this section I show you how to do this. There are actually many things that you can do with the IDE, so you might be surprised as you read this section.



If you are a language developer, you can add support for your language to the Visual Studio .NET. For example, if you have a Pascal compiler, you can create an add-in that lets people use Visual Studio .NET to develop programs in your Pascal language. To do this, use the Visual Studio Integrator Program (VSIP), which is a special program with Microsoft where you get additional SDKs for writing lower-level components for Visual Studio .NET. This program requires a separate agreement with Microsoft, meaning it is not a part of the Visual Studio .NET product that you normally purchase. For information on this program, visit <http://msdn.microsoft.com/vstudio/vsip/default.asp>.

Manipulating the Windows

Although most people who use Visual Studio .NET know how to find their way around the IDE, not everyone knows these interesting tidbits regarding how Visual Studio .NET handles its tool windows. In addition to knowing all the little tricks to getting around in Visual Studio .NET, it's also important to understand how the windows behave if you're writing macros and add-ins that might manipulate the windows.

First, Visual Studio .NET distinguishes between two different types of windows: *document* and *tool*. Document windows are those into which you type your code, after which their names appear in the Window menu. Tool windows are the auxiliary windows that do not hold code. Figure 1.1 shows an example of a document window; Figure 1.2 shows an example of a tool window.

```

Stat Page  Connect.cs
CSAddin1.Connect
namespace CSAddin1
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;
    using System.Runtime.InteropServices;
    using EnvDTE;

    [read me for Add-in installation and setup information.]
    /// <summary>
    /// The object for implementing an Add-in.
    /// </summary>
    [GuidAttribute("2807B82D-1595-4D4E-93C6-02263436DEB8"), ProgId("CSAddin1.Connect")]
    public class Connect : Object, Extensibility.IDTExtensibility2, IDTCommandTarget
    {
        /// <summary>
        /// Implements the constructor for the Add-in object.
        /// Place your initialization code within this method.
        /// </summary>
        public Connect()
        {
        }

        /// <summary>
        /// Implements the OnConnection method of the IDTExtensibility2 interface.
        /// Receives notification that the Add-in is being loaded.
        /// </summary>
        [space: none="application"]
        Root object of the host application.
        // ...
    }
}

```

Figure 1.1 A document window.

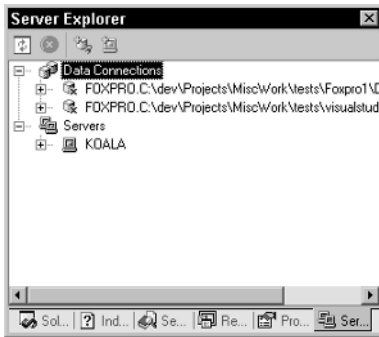


Figure 1.2 A tool window.

Visual Studio .NET also supports a tabbed window. There is a dedicated window (which I'll call a document frame) that contains tab sheets, each holding a document window. You can split the document frame into multiple smaller document frames, either vertically or horizontally, each with its own set of tabs. To do this, right-click on one of the tabs in the document frame and choose either *New Horizontal Tab Group* or *New Vertical Tab Group*. You can move document windows within a document frame by dragging one tab over another; this will change the *z-order* (that is, the front-to-back ordering). Or you can move a document window to another document frame by dragging the document window's tab onto any location on the new document frame. Figure 1.3 shows an example of the document window frame split to contain two documents side by side.

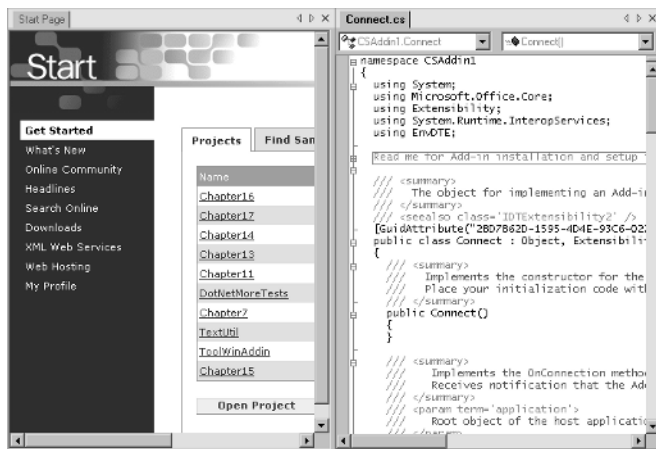


Figure 1.3 An example of a split document window.

Similarly, the tool windows can coexist alongside one another inside a tool frame window that has tabs. You can rearrange the z-order by dragging the tabs, and you can move a tool window from one tool frame to another by dragging the tab to the new tool frame. But unlike document windows, you can drag a tab and drop it so it gets its own floating tool frame window. To do this, grab the tab of your chosen tool window and drag it over the middle of another frame—*not* on a tab and not on the border of the frame. (If you drop it on a border, the window will move to another frame, rather than become floating.) The IDE will display a rectangular box showing where the frame will appear if you let go of the mouse button. But, remember, when you do this, you get a new tool frame. That means you can drag other tool windows onto this same tool frame, thereby either adding more tabs to the frame or splitting the frame either horizontally or vertically, putting one tool window in one half, and the other tool window in the other half. Figure 1.4 shows an example of such a split tool window.

To drop the tool window onto another tool window's frame so that the frame splits, drop the tool near one of the borders of the frame: near the left to split the frame horizontally, and drop your tool window on the left half; near the top to split the frame vertically, and drop your tool window on the top; and so on. (When the frame window splits like this, it really becomes two frame windows sharing a common outer window.)

But if you prefer to drop your tool window so it gets its own tab in the frame, rather than splitting the frame, drop the tool window either on the title bar of the tool frame or on the existing tabs of the frame, if there are any. (There won't be any tabs if the frame only holds a single tool window.) And speaking of which, if you need to drag a tool window and it's the only tool window in a frame, and therefore has no tabs, just drag the title bar of the frame. The IDE will interpret that as dragging the tool window.

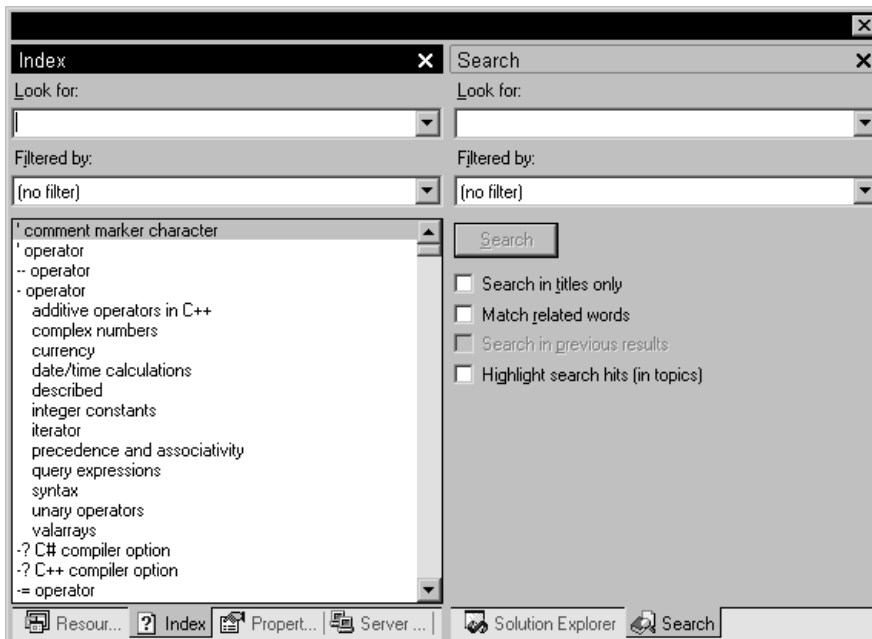


Figure 1.4 A split tool window.

Also be aware that you can turn off some of this behavior. If you want your tool window to be in its own floating frame where you cannot dock it, and the frame will not allow other tool windows to enter it, right-click on the tool window's tab, and in the popup menu choose Floating. Yes, it's possible that your tool window is *already* floating; but in this case, the word *means something different*: It means the tool window gets its own private frame, hence the IDE considers this form of floating the opposite of *dockable*. Until now, all the behavior I've been describing refers to tool windows that are dockable (yes, even when they're floating—but not floating as far as the IDE is concerned).

Now for the strange part: You can drop a tool window on a document frame and it will get a tab inside the frame, as shown in Figure 1.5. However, the document frame is still a document frame and behaves like one. But you cannot grab the tab of a document window and drop it on a tool frame, because documents cannot live inside a tool window.

But wait, there's more! You can also set up tool frames (not tool windows, but frames) so that they are in auto-hide mode. When in this mode, they appear as a small tab on the left or right side of the window. When you drag the mouse over one of these tabs, the window appears to pop out of the side (just like in one of those animal documentaries where the sand crab jumps out of the earth to snatch its prey).

To turn on the auto-hide mode, first dock the tool window or frame, if it's not already docked. Then right-click on either the title bar of the frame or one of the tabs (any tab will do, as long as it's on the same frame). In the popup menu that appears, choose Auto Hide.

Now remember, this auto-hide behavior works only for tool frames, not tool windows. And when you auto-hide a tool frame, it gets one side tab with multiple icons on it, one for each tool window, as shown in Figure 1.6. When you drag the mouse over the icon for one of the tool windows, that tool window will pop open and its name will appear beside the side tab. (There's also some strange widening behavior with the tab as the name appears and the other name disappears, but I'll let you explore that rather than try to describe it here.)

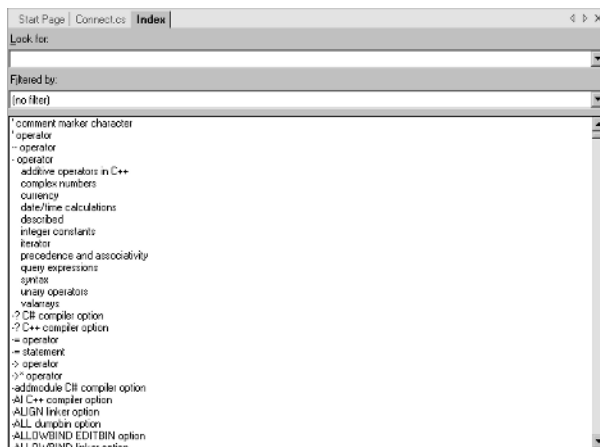


Figure 1.5 A tool window inside the document frame.

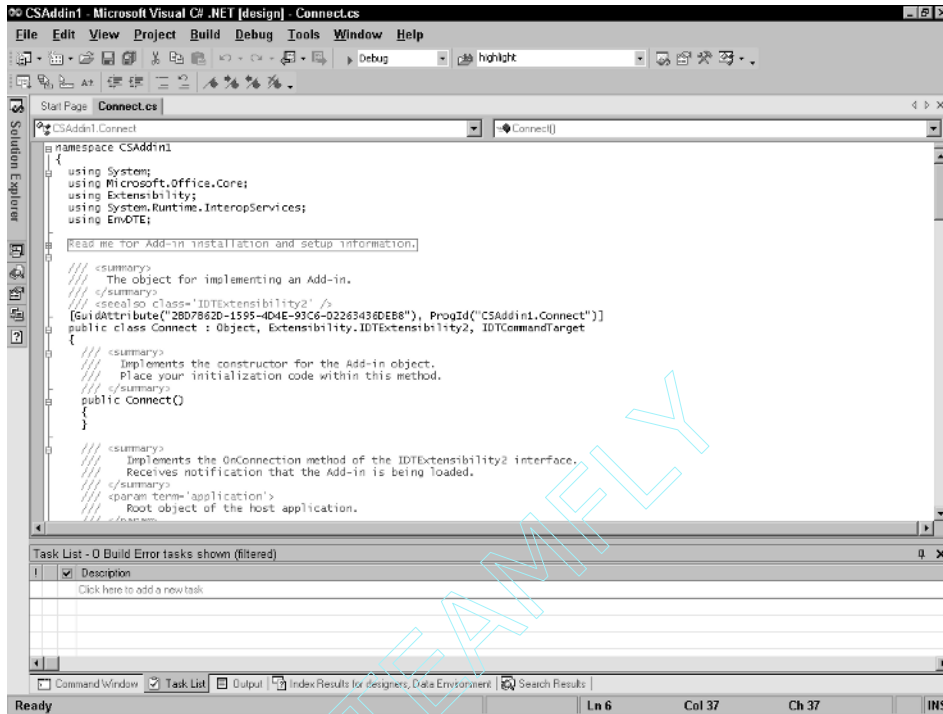


Figure 1.6 The tool window tabs are on the left.

Now let's assume you like the auto-hide behavior but don't like the way the tool windows share a tab. In this case, you can give each its own tab. To do that, first give the tool window its own frame (by dragging it around until it gets its own frame); then right-click the frame's title bar (there will be no tab to click) and choose Auto Hide.

When you write macros and add-ins, you have access to this windowing behavior. Here's a macro that moves the Server Explorer to a document window:

```
Public Sub ServerExplorerToDocument()
    Dim win As Window
    win = DTE.Windows.Item(Constants.vsWindowKindServerExplorer)
    If win.AutoHides = False Then
        win.IsFloating = False
    End If
    If win.Linkable = True Then
        win.AutoHides = False
        win.Linkable = False
        win.Visible = True
    End If
End Sub
```

This rather simple macro first allocates a variable called `win` of type `Window`. `Window` is a class in the automation engine that encapsulates the tool and document

windows in the IDE. (That said, be aware that the online help lists the Window class under Window Object, even though it is indeed a class.)

The second line initializes `win` with the Server Explorer window. To find the Server Explorer window, the macro first grabs the `DTE` instance (this time we're dealing with an object called `DTE`, not a class), then gets its `Windows` property, which is a container object that holds all the windows. The `Item` property is the actual container data, and you can access `Item` as a function. To find the Server Explorer window, you call the `Item` function, passing an enumerated data type called `vsWindowKindServerExplorer`. This cumbersome word is an item in the `Constants` object, and thus I fully qualify the name by starting it with `Constants`.

Next I check whether the `AutoHides` property is `True`. This property refers to the Auto Hide feature that I described a few moments ago; when it's true, it means the window has the Auto Hide feature enabled. If this property is `False`, then I set the `IsFloating` to `False`. Checking `AutoHides` before setting `IsFloating` to `False` took some calculating on my part to get the correct order for setting the properties. The reason I first checked `AutoHides` is that I wanted `IsFloating` to be `False`; but the problem is that if `AutoHides` is `True`, the macro engine issued a runtime error (in the form of a popup message called Invalid Parameter) when I tried to set `IsFloating` to `False`. That's because the Window class only lets you modify `IsFloating` if `AutoHides` is strictly `False`. Yes, it's kind of strange, but true.

Next I check whether `Linkable` is `True`. Why? Because if `Linkable` is `False`, it implies the window is already living as a Document window, and the macro can finish. But if it's `True`, I go ahead and reset the various properties. First, I set `AutoHides` to `False`, then `Linkable` to `False` (this is where the window actually switches to being a document window), and, finally, `Visible` to `True`, so you can actually see the window.

Here's a macro that moves the Server Explorer back to a tool window:

```
Public Sub ServerExplorerToTool()
    Dim win As Window
    win = DTE.Windows.Item(Constants.vsWindowKindServerExplorer)
    If win.AutoHides = False Then
        win.IsFloating = False
    End If
    If win.Linkable = False Then
        win.Linkable = True
        win.Visible = True
    End If
End Sub
```

This macro pretty much reverses the items in the previous macro, notice in this case I still go through the same rigmarole of setting `IsFloating` to `False`. I do this because even though I'm switching the window back to its own tool window, and even though it will appear to be floating, remember that to the IDE floating doesn't mean the same as it does to us. To reiterate, *floating* is the opposite as *dockable*. So even though I want the window to appear to be floating, I still want it dockable. Thus, I set `IsFloating` to `False`.

Then I check whether `Linkable` is `False`, because I only want to make the changes if the window is presently a document window. And if it's a document window, `Linkable` will be `False`. Then when I switch `Linkable` to `True`, this causes the window to switch back to being a tool window.

An interesting aspect of Visual Studio .NET is that you can choose whether you want your document windows to be tabbed windows or MDI windows (MDI stands for Multiple Document Interface). (This applies only to document windows, not tool windows.) When you open up the Options dialog through `Tools` → `Options`, under the Environment category, the General options allows you to choose between Tabbed Documents and MDI environment. (Be aware that if you change this setting, you will have to restart Visual Studio .NET before you will see the changes.) Figure 1.7 shows what the Visual Studio .NET IDE looks like with MDI windows.

If you choose Tabbed Documents, your document windows will have tabs at the top, allowing you to choose which document you wish to edit. Additionally, the upper-right corner of the document window will have a small box with an `x` in it that you can click to close the current document. (The document window itself will not close unless there is only one document in the window and you close the document.) To the left of the close box are two arrows (that look like triangles) that you use to scroll the tabs into view if there are too many tabs to fit in the view.

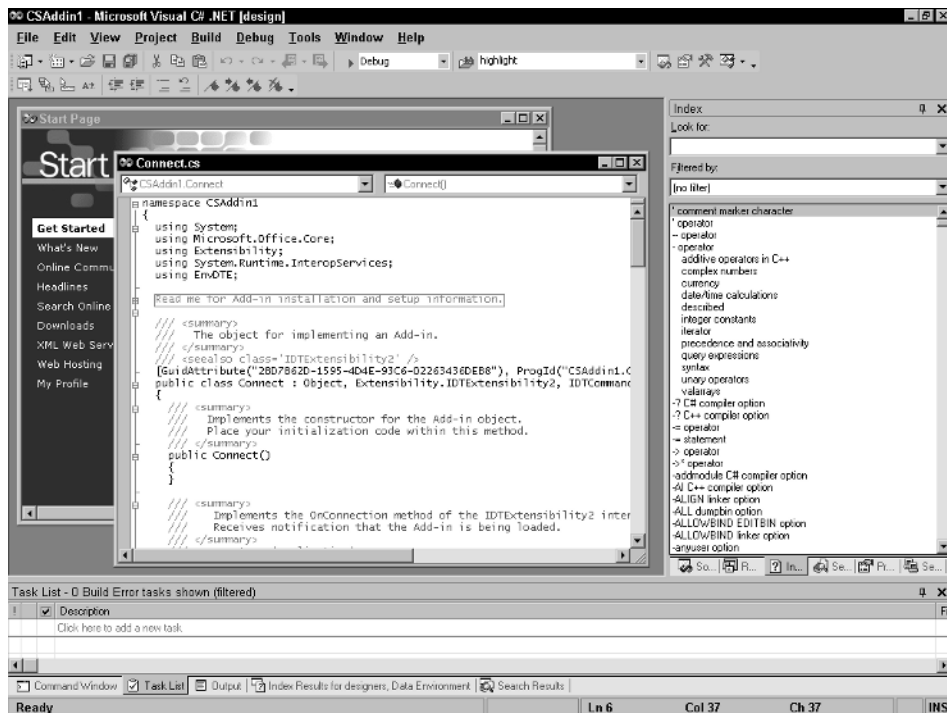


Figure 1.7 The IDE with MDI windows.



Be aware of an odd behavior in Visual Studio .NET that takes some getting used to: When you are in Tab Documents mode and you are looking at a window, you can press Ctrl+Tab to switch to the next window. But “next” does not necessarily imply the window whose tab is directly to the right of the current window’s tab, since the IDE maintains a z-order (remember, that’s a front-to-back order) that is independent of the order of the tabs. Note, also, that if you want to go in the opposite direction, back to front, you can use Shift+Ctrl+Tab.

The alternative to Tabbed Documents is the MDI environment. MDI is the standard interface for a lot of older Windows programs. When you choose this option, the tabs are not present in the document windows, and each document gets its own window. You can maximize, minimize, and restore the document windows inside the IDE. When they are restored (but not maximized), they have their own title bar with the usual goodies—the title, a minimize button, a maximize/restore button, and a close button. When you maximize the window, it takes up the entire free area of the IDE (the area not used by the tool windows), but it does not have its own title bar; instead, the minimize, maximize/restore, and close buttons appear to the right of the main menu bar in the IDE.



Most people prefer the newer tabbed look, simply because the names of the documents are all right there in the form of tabs. It’s easy to quickly switch between them. With MDI, you can’t see all your document names at once until you click on the Window menu to get a list of them.

Managing the Toolbars and Commands

Toolbars aren’t nearly as complicated to describe as the tool and document windows. The toolbars live around the edges of the IDE window. When they’re docked at the top or bottom of the IDE window, you will see a small dashed bar to the left. You can drag this bar to move the toolbar around. You can slide it around, swap it with another toolbar in the same docking edge of the IDE window, or move it to another edge. And although the dashed bar’s purpose is for dragging, you can also drag the toolbar from any *spacer* between the buttons and controls on the toolbar. (Spacers are areas that are not clickable; they simply provide for space between the controls, and they have a vertical line when the toolbar is docked in the top or bottom edge, or a horizontal line when the toolbar is docked on the left or right edge.)

Toolbars can also be floating; if you drag the toolbar to somewhere in the middle of the IDE’s window and let go, the toolbar will get its own window, with its own title bar. The title bar is smaller than a standard Windows title bar, and it contains the name of the toolbar and a close button.

You can add and remove toolbars by right-clicking in the free space around the toolbars or by right-clicking on the dashed line of the toolbar or on the spacers. When you right-click, you will see a popup menu listing all the toolbars available. Those that are visible have checkmarks by them. You can hide a toolbar by unchecking its name in the popup menu, or make the toolbar visible by checking its name.

The final item in the popup menu is *Customize*. When you click this item, you will see the *Customize* dialog box. (This is the same dialog box that appears when you choose *Tools*→*Customize*.) In this dialog are three tabs: *Toolbars*, *Commands*, and *Options*.

The *Toolbars* tab, shown in Figure 1.8, lists all the toolbars in a checked listbox. The visible toolbars have checkmarks by them. This tab is handy, as you can add new toolbars. In this tab is a button called *New*, and when you click it, a small window opens asking you for the name of the new toolbar. When you type the name and click *OK*, the new toolbar will appear floating somewhere in the middle of the IDE window, with no buttons inside it. Its name will also appear in the checked listbox, with a check beside it. To add and remove buttons from it, use the *Commands* tab.

The *Commands* tab, shown in Figure 1.9, is the second tab in the *Customize* dialog box. It lists all the commands available. These include all the menu items and the macros, divided up into categories. The left listbox gives the categories, and the right listbox gives the commands for that category. If there's a command you wish to add to a toolbar, find the command in listbox on the right, then drag the command's name to the toolbar on which you want to put the command. You can do this for any toolbar, including any toolbars you created. To remove a command from a toolbar, just drag the button off the toolbar and let go anywhere. (But, note, if you drag it onto another toolbar, you will move it to the other toolbar.)



Figure 1.8 The Toolbars tab of the *Customize* dialog box.

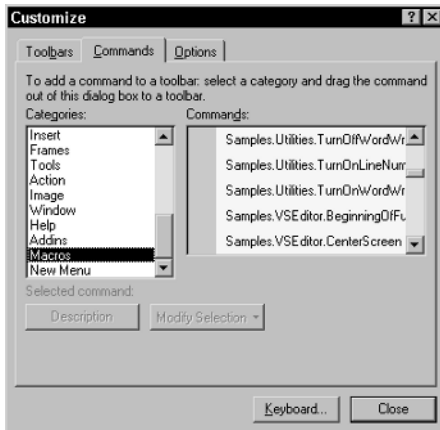


Figure 1.9 The Commands tab.

There's another way you can add and remove commands from a toolbar. (Once again, we're bordering on feature overload with this product.) While the toolbar is docked on the top or bottom of the IDE, you will see at the right end a little arrow pointing down; or, if the toolbar is docked on the right or left side, the arrow will be at the bottom and pointing left; finally, if the toolbar is floating, you will see this arrow on the right-hand side of the toolbar's title bar, to the left of its close button. When you click the arrow, you will see a drop-down menu with one item on it called Add or Remove Buttons. When you move your mouse over the item, you will see a submenu appear with either one or two items. If the toolbar is one of the standard toolbars (that is, not one you created) the first menu item will be the name of the toolbar, along with the commands relating to that name. (For example, the Web toolbar has a Web menu item containing commands related to the Web and browser.) These commands will have checkboxes next to them, and you can check and uncheck them to add or remove the commands from the toolbar. (Yes, it's true: These are menu items, but they have clickable checkboxes next to them.) The second menu item is Customize, which brings up the standard Customize dialog box. If the toolbar is a custom toolbar, then you will only see the Customize menu item.

The Options tab is the third in the Customize dialog box. When you click on this tab, the first thing you'll notice is that the top half of the controls are disabled. That's because the toolbars section of the IDE come from Microsoft Office XP, and with that section came this Customize dialog box. And as it happens, Microsoft decided the feature called Personalized Menus and Toolbars wouldn't be made available in the Visual Studio .NET IDE; so instead of removing them altogether, the company simply disabled them. But the bottom half *is* available, and these options are:

Large Icons. This one is self-explanatory. Selecting it gives you large icons on your toolbars. However, it does not substitute larger, higher-resolution icons for the existing icons. Instead, it just magnifies the existing ones.

List font names in their font. When you have a font list on a toolbar, the font names will be in their own font.

Show ScreenTips on toolbars. When this is selected, a description will appear in a tooltip window when you float the mouse over a command on a toolbar.

Show Shortcut keys in ScreenTips. If you've assigned a shortcut key (also called a *hotkey*) to your commands, the tooltip will also include the shortcut key.

Menu animations. Supposedly, this will change the way the menus appear. Unfortunately, it doesn't seem to work. (Or, probably, Microsoft didn't implement it when it reused the Customize dialog box.)

Keeping in mind that this book is about macros and add-ins, not about how to use the IDE, take a look at the list of Commands. There you will see all the commands currently available to the IDE. You can call any of these commands from your macros and add-ins. In a moment, I'll show you a macro that lists all the commands. But before you can run this macro (and after you type it in or, at least, after you have created the module where you'll be typing it), you need to do the following:

1. Make sure you're in the Macros IDE and that your macro project is open.
2. Choose Project⇒Add Reference. The Add Reference dialog box will open (see Figure 1.10).
3. In the Add Reference dialog box that opens, scroll down and find System.Drawing.dll. Click on it, then on the Select button. You will see it added to the Selected Components list at the bottom of the dialog box. Click OK.
4. In the code editor, scroll to the top of your module and add the following line immediately after the existing line `Imports System.Diagnostics`:

```
Imports System.Windows.Forms
```

The reason you need to add the reference is that the macro makes use of a sizing class called `ClientWidth` (to set the listbox's size to match that of the form), and this class lives inside the `System.Drawing` DLL (which is actually an assembly). (I talk about assemblies in Chapter 5, "Just Enough .NET Architecture.") The reason for the additional `Imports` statement is in the code I make use of various objects in the `System.Windows.Forms` assembly, but I don't fully qualify the names. The `Imports` statement lets me type the class name without preceding it with `System.Windows.Forms`.

Now here's the code for the macro:

```
Sub GetAllCommands()  
    Dim cmdlist As Commands = DTE.Commands  
    Dim c As Command  
    Dim myform As Form = New Form()  
    Dim list As ListBox = New ListBox()  
    Dim i As Integer = 0  
    Dim count As Integer = cmdlist.Count  
    Dim status As EnvDTE.StatusBar = DTE.StatusBar  
    list.Left = 0
```

```

list.Top = 0
list.Width = myform.ClientRectangle.Width
list.Height = myform.ClientRectangle.Height - 15
list.Anchor = AnchorStyles.Bottom + AnchorStyles.Top + _
    AnchorStyles.Left + AnchorStyles.Right
myform.Controls.Add(list)
For Each c In cmdlist
    status.Progress(True, "", i, count)
    If c.Name <> Nothing Then
        list.Items.Add(c.Name)
    End If
    i = i + 1
Next
myform.ShowDialog()
status.Progress(False)
End Sub

```

The first seven lines of this macro (the lines that start with Dim) declare the local variables for the macro's subroutine. The first of these is `cmdlist`, which the macro initializes to `DTE.Commands`. The `DTE.Commands` object is the key to this macro, as it contains a list of all the commands. The list lives in `DTE.Commands` object's `List` property. The second variable, `c`, is a holder that's used later on when the macro loops through the commands; this variable holds the current command in the list.

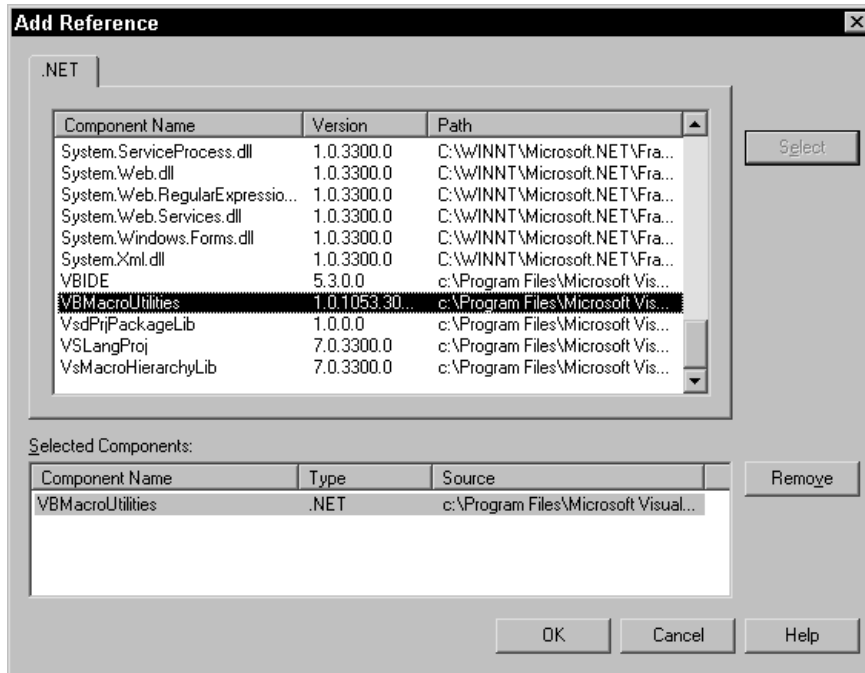


Figure 1.10 The Add Reference dialog box for the Macros IDE.

The third variable creates a new instance of the `Form` class, which is the macro name for a window. The next line creates a `ListBox` instance. Note that this listbox doesn't automatically get put inside the window until later on when I call `Controls.Add(list)` for the `Form` object.

The next two variables, `i` and `count`, are used for looping through the command list. Notice that I'm initializing the count with the command list's `Count` property, which is the number of commands in the list.

The variable that follows, `status`, is a reference to the main status bar in the Visual Studio .NET IDE's main window. When the macro cycles through the command list, the macro puts up a progress bar in the status bar to show how far along it is in gathering up the list. (Gathering the list takes a few moments to do; that's why I decided to use a status bar.) The next four lines set the size of the listbox; the following line *anchors* it, which simply means that when you resize the form, the listbox will resize with it. Finally, the line to add the listbox control to the form follows.

Next begins the retrieval of the commands. This loop cycles through the command list and updates the progress bar accordingly; it then adds the commands to the listbox. (I first make sure the command isn't an empty string, since you don't want to add any empty strings to the list.) Finally, after the loop finishes, the macro shows the window in the form of a dialog box by calling the form's `ShowDialog` function.

To clean things up after the user closes the window, the macro clears out the progress bar (otherwise it will stay there).

Customizing the Menus

The menus in the Visual Studio .NET IDE are fully customizable. First, the Tools menu has its own customization feature that is pretty handy for launching external programs and scripts.

For example, for a project I recently completed, I wrote a VBScript program that runs under the Windows Shell system that automatically does a screen capture of the Visual Studio .NET IDE and saves it to a file. Then I added an item under the Tools menu that launches the script, passing a filename as a parameter.

Though this book isn't about VBScript, I do talk a bit about the Windows Script Host in Chapter 15. Here's the script I used for my recent project, if you're curious:

```
dim filename
Set objArgs = WScript.Arguments
if objArgs.Count = 0 then
    WScript.Echo "Error; No filename given."
else
    filename = objArgs(0)
    Set WshShell = WScript.CreateObject("WScript.Shell")
    WshShell.Run "PHOTOED.EXE.lnk"
    WScript.Sleep 200
    WshShell.AppActivate "Microsoft Photo Editor"
    WScript.Sleep 300
```

```

WshShell.SendKeys " %en%fa+{end}" '
    Paste, File-SaveAs, Select filename
WshShell.SendKeys filename ' Enter filename
WshShell.SendKeys "%tt" ' Select TIF type
WshShell.SendKeys "%s" ' Save it!
WshShell.SendKeys " %fc" ' Close it!
end if

```

This program went inside a file called `screencapture.vbs`. If you build a script such as this, you can then add a menu item under the Tools menu that accesses the script. To add the menu item, choose Tools→External Tools. In the External Tools dialog box that opens, click Add. The dialog box will add a new item to the Menu Contents listbox, and the Title edit control will be active. You then type the name of your menu item name into the Title control. (For mine, I typed “screenshot.”)

For the Command edit control, you type the name of the command you want to run. Make sure you include the `.exe` filename extension or other executable extension; otherwise, the IDE will not find the command. I usually also include the full path. Also, don’t put any command-line arguments here; they go in the next box. If you put them here, the IDE will get confused and won’t be able to find the program. To run my screenshot script, I typed “`c:\winnt\system32\wscript.exe`” into the Command edit control.”

Next, in the Arguments edit control, you list the arguments to your program. For my script, this is the script file itself, followed by the command-line arguments that get passed into the script. Here’s what I put for the Arguments edit control:

```

C:\Tools\screencapture.vbs c:\$(ItemFileName)

```

Now notice the variable `ItemFileName`. To fill that in, you can click the small arrow to the right of the Arguments edit control. This opens a popup menu containing a list of possible variables for the arguments. I chose Item File Name, and the dialog box automatically filled in the `$(ItemFileName)` variable for me.

Here are the variables available to you in the Arguments edit box:

Item Path, Item Directory, Item File Name, and Item Extension. These enter the name of the current document (typically a source code filename). The first, Item Path, refers to the entire path and filename, such as `c:\dev\system\main.cpp`. The second, Item Directory, refers to just the path name, up to and including the final backslash, but excluding the filename. The third, Item File Name, is just the filename, without the extension. The final, Item Extension is simply the filename extension.

Current Line, Current Column, Current Text. The first, Current Line, is the current line number where the cursor is presently located inside the document window. The second, Current Column, is the column number where the cursor is located. The third, Current Text, is the currently highlighted text in the document window. If there is no text highlighted, this will be an empty string.

Target Path, Target Directory, Target Name, Target Extension. These all refer to the path and filename of the target, which is the item you are building, such as `c:\dev\system\debug\MyProgram.exe`. This filename is specified in the current solution.

Project Directory, Project File Name. The first is the directory only of the project file, up to and including the final backslash, but excluding the filename. The second is the filename of the current project file only, without the path. (If you need both, you just enter one after the other.)

Solution Directory, Solution File Name. The first is the directory only of the solution file, up to and including the final backslash. The second is the filename only of the solution file.

For the Initial Directory edit box, you can enter the directory in which you want the program to start running. For my script, I left the initial directory blank. However, there are some variables here that you can enter. These comprise a subset of those in the list I just described. The variables available are Item Directory, Target Directory, Target Name, Project Directory, and Solution Directory.

After you enter the information into the External Tools dialog box, you will have a new menu item under the Tools menu.

There's another way you can modify the menus, which is somewhat different from most programs that allow customization. If you choose `Tools` → `Customize`, you will see the `Customize` dialog box. While this box is open, you can click on any menu item and, without letting go of the mouse, drag the menu item to another position—to within the same menu, to the top of the menu bar, or to any toolbar. In this manner, you can completely rearrange the menu items to your liking. (Of course, if you're using a computer that others use as well, and you change things around too much, you might have some unhappy coworkers.)

Also note that inside the `Customization` dialog box, if you click on the `Commands` tab and then scroll down in the `Categories` list box to the bottom, you will see the category `New Menu`. When you click this item, you will see a single item appear in the `Commands` list, called `New Menu`. This represents a new drop-down menu, and you can drag it to any of the open toolbars or to the main menu bar; or you can insert it into an existing menu by holding the mouse over the word on the main menu bar until the menu opens, and then drag the `New Menu` item down into the menu bar, as shown in [Figure 1.11](#).

Once you let go of the mouse, you will have a new menu. You can then change the caption on the menu by right-clicking the menu, and in the resulting popup menu clicking the `Name` item. The menu item itself has an edit control in it (it's quite fancy, as you can see), in which you can type the new name for the menu item.

Remember, you created an entire menu, not just a menu item. Technically speaking you created a menu item with a submenu. Thus, if your new menu is on the main menu bar, it will have a drop-down menu on which you can drop commands. If your new menu is on an existing menu, then you will get a secondary menu. Or, if you

dropped the new menu on a toolbar, the resulting item will have a small arrow that you can click to open a drop-down menu.

Then you can drag any command from the Commands tab in the Customize dialog box onto your new menu. This, of course, includes any of your own macros. Thus, you can create an entire drop-down menu on the main menu bar for your own macros, if you want. For example, on my system, I used the New Menu item in the Commands list to add a new menu on the main menu bar, called Macros. Then, under the Commands tab of the Customize dialog box, I chose the Macros category. From the Commands list I found the macros that I wrote and dragged them to the drop-down area of my new Macros menu. Finally, with the Customize dialog box still open, I right-clicked on each menu item and used the Name item to give each macro a friendlier name. Thus, I had quick menu access to all my macros.



Interestingly, in the Macros IDE, you also have access to a Customize dialog box, through which you can modify the menus and toolbars inside the Macros IDE itself. There are fewer commands available to you, however, as you only have access to those that pertain to the macros.

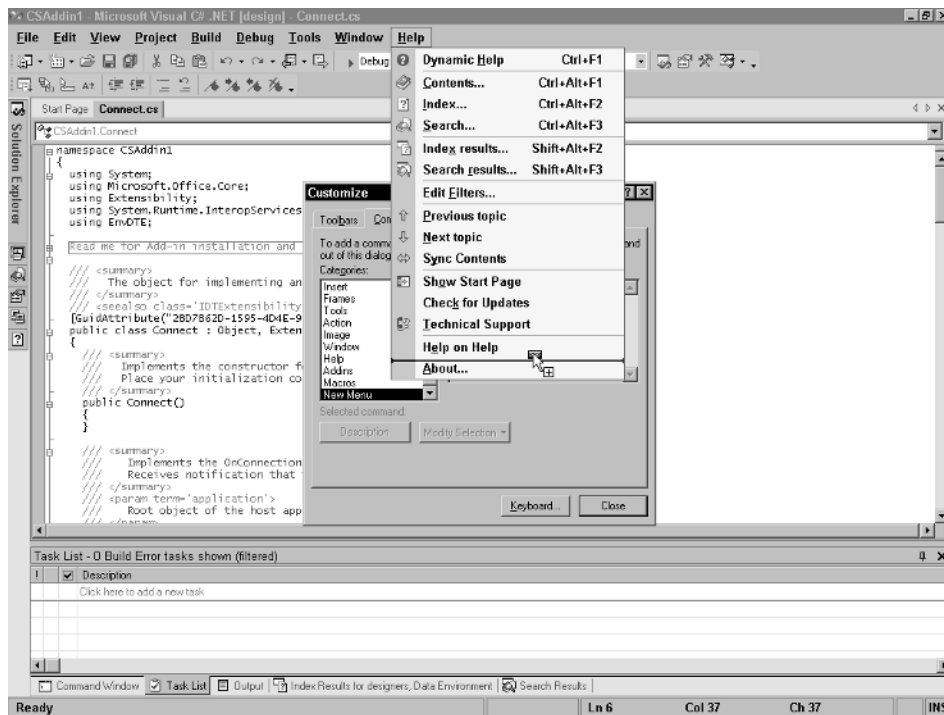


Figure 1.11 A new menu.

Moving Forward

In this chapter, I gave you a feel for what macros and add-ins are and where they fit into the bigger picture of Visual Studio .NET. I described ways you can configure the Visual Studio .NET IDE, and I gave you a few macros to get started. This let you experience a bit of the Macros IDE, which, as you could see, is a separate IDE that looks very much like the main IDE.

As you saw in the macros that you wrote, the language for macros is VB.NET. Therefore, I'm devoting the next chapter, Chapter 2, "Just Enough VB.NET," to a lesson in VB.NET. In that chapter I take you through the VB.NET language, showing you its syntax and interesting features.

Later in the book, as I get into add-ins, you will see that you can write add-ins in one of many languages, and for many of the examples I choose VB.NET. This, again, is the same version of Visual Basic that you use for macro programming. If you're not familiar with VB.NET, I recommend reading Chapter 2.

Just Enough VB.NET

The Visual Studio .NET macro engine uses VB.NET as its macro language. This means you need to know a bit of VB.NET to program macros. Entire books have been written about this language, but rest assured, you don't need to be an expert in it to write macros. Instead, with just a bit of an introduction, you can learn the nuts and bolts of VB.NET simply by doing. Probably, by virtue of the fact that you're reading this book, you're most likely a programmer already; otherwise you would probably have little interest in working with Visual Studio .NET, much less enhancing it with macros and add-ins. So in this chapter I'm give you just enough information about VB.NET to get you going. It comprises subsections on various important topics, along with numerous examples so you can more quickly become familiar with the language. I'm also assuming you are familiar with at least one other programming language, such as C++ or C#, and that you have a basic familiarity with variables, control structures, and classes and objects.

First, here are some general thoughts about VB.NET:

- VB.NET, like other versions of BASIC, is not case-sensitive. You can type the word `MSGBOX` or the word `MsgBox`. That said, note that the editor has a preferred case for most keywords and identifier names, and so will correct your casing for you after you finish typing in a line of VB.NET code.

- The macro editor includes a great deal of automatic formatting in addition to setting the case. You normally don't have to worry about the indentations of your code, because the macro editor will automatically set the indentations for you. (If you don't like this feature, you can turn it off. To do so, make sure you are inside the Macro IDE, not the main IDE, and choose Tools→Options. The Options dialog will open, as shown in Figure 2.1. In the treeview on the left, choose Text Editor, then Basic. In the Visual Basic-specific Options on the right, uncheck *Pretty listing (reformatting) of code*.)
- At times, the macro editor also automatically enters text for you. Specifically, when you enter a block statement such as a subroutine or a block-`IF` statement, the editor automatically adds the closing line of the block for you. (You can also turn this feature off if you don't like it. In the same options screen just described, uncheck *Automatic insertion of end constructs*.)



If you find that you enjoy working in the VB.NET language but are unsure whether to use it to develop applications besides simple macros, be assured that Visual Studio .NET compiles VB.NET source code to the same Microsoft Intermediate Language (MSIL) to which it compiles C# and C++ managed applications. Therefore, in terms of performance, there will be no difference between a program written in VB.NET and C++ with managed extensions.

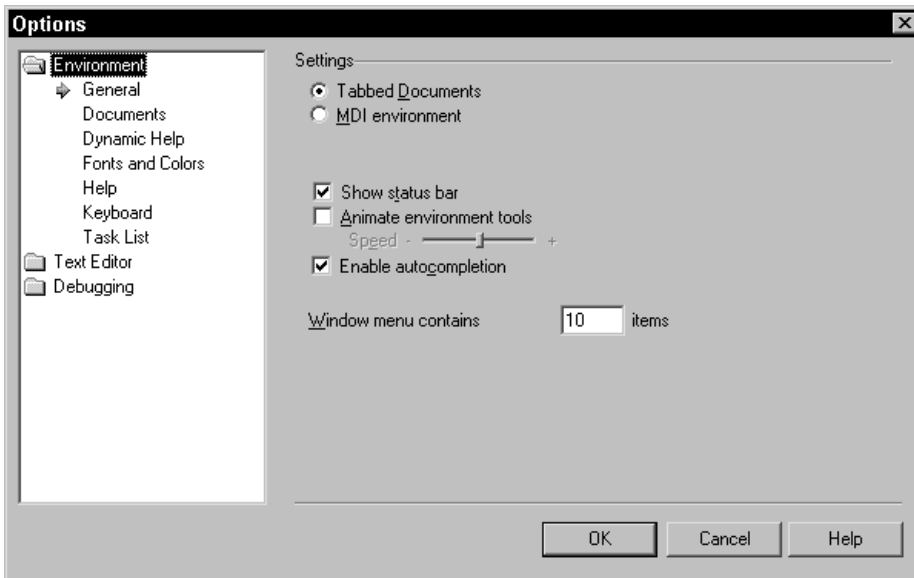


Figure 2.1 The Options dialog for the Macros IDE.

VB.NET Subroutines and Functions

Like most programming languages, VB.NET allows you to divide your code up into procedures and functions. In the Visual Basic world, however, procedures are called *subroutines*. The primary difference between a subroutine and a function is that a function has a return value, whereas a subroutine does not. (If you're coming from a C++ world, then think of a subroutine as a function with return type void.)

Here's some sample code showing the format of the subroutine in VB.NET:

```
Sub MySubroutine()  
    MsgBox("Hello world")  
End Sub
```

The first line starts with the keyword `sub`, then declares the name of the subroutine, and finally lists the parameters inside parentheses. This example has no parameters. The lines that follow are the code for the subroutine. The subroutine ends with the keywords `End Sub`.

Now here's an example of a subroutine that uses parameters:

```
Sub MySubroutineWithParams(ByVal a As Integer, ByVal s As String)  
    MsgBox(a)  
    MsgBox(s)  
End Sub
```

This subroutine has two parameters, an `Integer` and a `String`. Each of them is passed by value (thus the `ByVal` keyword preceding each), which means the subroutine receives a copy of the arguments passed in and cannot modify the originals.

If you would like your subroutine to modify the originals, use the `ByRef` keyword in place of the `ByVal` keyword:

```
Sub MySubroutineWithParams(ByRef a As Integer, ByVal s As String)  
    MsgBox(a)  
    MsgBox(s)  
    a = a + 1  
End Sub
```

This subroutine calls the preceding subroutine and shows that the value passed in did indeed change:

```
Sub TestSub()  
    Dim num As Integer = 10  
    MySubroutineWithParams(num, "Hello")  
    MsgBox(num)  
End Sub
```


Now here's an example of a function:

```
Function Cube(ByVal num As Integer) As Integer
    Return num * num * num
End Function
```

You can also use `ByRef` in your function, although most people usually consider that bad practice, since a function's primary purpose is to simply return a value, not modify the parameters passed into it. Nevertheless, the option is there if you need it.

There are two ways you can return a value from a function: The previous code used the `Return` statement; the alternative is to set the name of the function equal to the value you are returning. Here's the same function using this alternate method:

```
Function Cube(ByVal num As Integer) As Integer
    Cube = num * num * num
End Function
```



When you write your macros, you will write them as subroutines with no parameters. If you have a subroutine that takes parameters, or a function, it will not show up in the Macro Explorer in the main IDE. However, you can still use subroutines with parameters and functions in your code; your macro subroutines can call these other subroutines and functions.

VB.NET Variables

In VB.NET you normally declare your variables before you use them, as you do in C++ and other languages. To declare a variable, you use a dimension statement such as this:

```
Dim mystr As String
```

This declares a variable of type `String` called `mystr`. You would, for example, put the declaration at the beginning of a subroutine, as here:

```
Sub Variables()
    Dim mystr As String
    mystr = "Hello"
    MsgBox(mystr)
End Sub
```

The second line stores a string inside the string variable. Notice that in VB.NET strings require double quotes.

The `MsgBox` routine displays a message box containing the text passed into the call to `MsgBox`. If you type this subroutine into a macro module, you will see the name `Variables` under the module name in the `Macros explorer` within the main IDE. If you then double-click the name `Variables`, the macro will run, resulting in a message box opening up, with the word "Hello" in it.

You can also initialize the variable in the dimension statement by following the type name with an equal sign and an initial value, like so:

```
Sub Variables()  
    Dim mystr As String = "Hello"  
    MsgBox(mystr)  
End Sub
```

To declare several variables at once, you can string them together into a single statement, like so:

```
Dim a1, a2 As String
```

However, when you declare multiple variables on a single line, you cannot initialize them with an equal sign and a value on the same line as the declaration. Instead, you have to initialize them separately:

```
Dim a1, a2 As String  
a1 = "Hello"  
a2 = "there"
```

The VB.NET language has several built-in types. You can declare a variable to be of any of these built-in types; you can declare it to be of any of the .NET framework classes; finally, you can declare it to be one of the macro objects, such as `EnvDTE.StatusBar`.

Each built-in type has an associated .NET class. For instance, the .NET class for the built-in type `Integer` is `Int32`. Thus, when you declare a variable as one of the built-in types, the variable has a set of member variables and member functions that you would normally expect to find only in an object. For example, the .NET class `Int32` has a member function called `ToString`, which allows you to convert the integer to a string using a format specifier. (If you're familiar with `printf` format specifiers, .NET format specifiers have no similarities to `printf` specifiers.) For example, this code:

```
Dim n As Integer  
Dim s As String  
n = 10  
s = n.ToString("C")  
MsgBox(s)
```

saves the integer 10 in the variable `n`, then calls the `n` variable's `ToString` function, just as if `n` were an object, not a built-in type. (In fact, it really *is* an object.) This particular `ToString` example formats the number as currency (that's what the `C` stands for). Thus, when the message box opens, it will show the string "\$10.00". (If you're interested in exploring the format specifiers, open up the Visual Studio .NET online help,

and go to the contents. From there, drill down as follows: Visual Studio .NET↔.NET Framework↔Programming with the .NET Framework↔Working with Base Types↔Formatting Types. (This section contains a complete description of the format specifiers and how to use them.)

Here's a list of the built-in types, along with their associated .NET classes and a description. Note that *signed* means an integer variable can hold a negative number, positive number, or 0. *Unsigned* means the integer variable can hold only positive numbers or 0. Note also that although I'm listing the .NET classes here, the primary use for the class names is to look up the member functions and variables for the class in the online help. In your macro programming, you will not use the .NET class names.

Byte (class Byte). This is an 8-bit unsigned integer. Its possible values, therefore, range from 0 to 255.

Short (class Int16). This is a 16-bit signed integer. Its possible values range from -32768 to 32767.

Integer (class Int32). This is the most commonly used integer type. It's a 32-bit signed integer, ranging from -2,147,483,648 to 2,147,483,647.

Long (class Int64). This is an integer with double the precision as the standard Integer type, allowing for enormously large positive or negative numbers. If you're curious, the range is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. (Yes, that's nine *quintillion*.)

Single (class Single). This is a 32-bit floating-point number. Its range is -3.402823e38 to +3.402823e38, although its precision is limited to 32 bits.

Double (class Double). This is a 64-bit floating-point number, ranging from -1.79769313486232e308 to +1.79769313486232e308.

Boolean (class Boolean). Variables of this type hold only one of two values: True or False.

Char (class Char). This is a type representing a single character. Note that in .NET, the Char type is a wide character, meaning it takes up 2 bytes, not just 1 as in most character types on other platforms.

Decimal (class Decimal). This is a unique integer type that takes up 96 bits (that is, 12 bytes). It can hold integers in the range (get ready for this) -79228162514264337593543950335 to 79228162514264337593543950335. This type is primarily useful in financial and accounting applications where large numbers and no rounding errors are required.

Object (class Object). This is simply the root class from which all .NET objects are derived.

String (class String). This is the standard .NET string type. Refer to the section titled "Strings in VB.NET" in this chapter for more information.

Date (class Date). This is a flexible data type for handling dates.

One particularly interesting item of note about the two floating-point types (`Single` and `Double`) is that they can hold any number in their respective range; or they can

hold the value `PositiveInfinity` to represent positive infinity, `NegativeInfinity` to represent negative infinity, or `NaN` meaning “not a number.” If you divide a positive number by 0 (according to .NET anyway) you will get `PositiveInfinity`, like so:

```
Sub TryInfinity()
    Dim f As Single
    f = 1.0 / 0.0
    MsgBox(f)
End Sub
```

When you run this macro, you will see a message box with the word “Infinity.” To get negative infinity, you divide a negative number by 0. To get “NaN,” you divide 0 by 0.

The `Decimal` type is a bit strange to use. To store a number in a variable of type `Decimal`, you can either use a string, as in this code:

```
Sub TryDecimal()
    Dim d As Decimal = "1111111111222222222233"
    MsgBox(d * 2)
End Sub
```

Or you can append a D on the end, like so:

```
Sub TryDecimal()
    Dim d As Decimal = 1111111111222222222233D
    MsgBox(d * 2)
End Sub
```

When you run this code, you will see a message box showing `222222222444444444466`. With the `Decimal` type, you can also apply a *fixed decimal point*. This is not like a floating-point number, which can lose accuracy in accounting packages. Instead, `Decimal` uses a fixed decimal point, which maintains accuracy by storing the digits as a whole number and keeps track of the decimal point’s position. (It’s the same idea as saying, “To add \$15.35 and \$2.86 on a calculator, just type in `1535 + 286`,” understanding that there’s an implied decimal to the left of the second digit.) Thus, you can try something like this:

```
Sub TryDecimal()
    Dim d As Decimal = 11111111112222222222.33D
    MsgBox(d * 2)
End Sub
```

When you declare a variable as a macro object, make sure that when you dimension a variable, you declare it as a type; don’t use a value in place of the type. Look at this line of code:

```
Dim b As StatusBar = DTE.StatusBar
```

It declares `b` to be of type `StatusBar`, and it initially sets it to `DTE.StatusBar`. `StatusBar` is a class, and `DTE.StatusBar` is an instance of `StatusBar`. Remember, in this book, you are writing macros, so most of the objects you access will be inside the IDE. Some of these objects might be the only instance of a class. For example, the IDE has only one status bar. To access this status bar, you access the `DTE.StatusBar` object. The easiest way to figure out the type, then, is to look up in the online help the name of the object you are trying to access. (Most of the objects have the usual names, such as `StatusBar`.) The online help will list the entry as an object (as in “`StatusBar Object`”). This online help entry looks as if it’s discussing a class (since it includes properties, methods, and events for the object). However, the entry is actually for the actual object. Now look down at the example (each object has one) and you’ll see how the variable is dimensioned. In the case of the status bar, the variable is dimensioned as type `StatusBar`.



In Chapter 3, “Introducing the Visual Studio Macro IDE,” I discuss the different macro objects available, along with many of the .NET classes that are useful to macros and add-ins.

You can declare constants as well as variables. Constants have fixed values, and once you declare them, you cannot change them. For constants, you use the keyword `const` instead of `dim`. Here’s an example:

```
Const company As String = "Macros, Inc."
```

Arrays in VB.NET

Arrays in VB.NET (and C++.NET and C# as well) are handled drastically differently from arrays in non-.NET languages. The reason for the difference is that the .NET framework includes an array class, and to get the array class working with the .NET languages, Microsoft totally integrated the .NET array class to the languages. Thus, VB.NET has an extremely powerful built-in array type.

One unusual aspect of VB.NET arrays is that the indices start at 0 and go up to, and include, the number you use when you declare the array. So if you declare an array of 100, the smallest index is 0 and the highest index is 100 (so really the size is 101).

The first unusual thing about the array type is that you don’t *have to* specify a size when you declare an array variable. Here are some examples. First, a simple array where I declare the size up front:

```
Sub ArrayTest()  
    Dim myarray(5) As Integer  
    Dim i As Integer  
    For i = 0 To 5  
        myarray(i) = i * 2  
    Next  
End Sub
```

Next is a simple two-dimensional array:

```
Sub ArrayTest()  
    Dim myarray(5, 3) As Integer  
    Dim x, y As Integer  
    For x = 0 To 5  
        For y = 0 To 3  
            myarray(x, y) = x * y  
        Next  
    Next  
End Sub
```

And now here's an example where I don't declare the size up front:

```
Sub ArrayTest()  
    Dim myarray() As Integer  
    Dim x As Integer  
  
    myarray = New Integer(5) {}  
    For x = 0 To 5  
        myarray(x) = x * 2  
    Next  
    MsgBox(myarray.Length)  
  
    myarray = New Integer(10) {}  
    For x = 0 To 10  
        myarray(x) = x * 5  
    Next  
    MsgBox(myarray.Length)  
End Sub
```

The first line of this code declares `myarray` as an array of integers, but it does not yet specify the size. In fact, the reason the line doesn't specify a size is that no array has been created yet. The third line, `myarray = New Integer(5) {}`, creates the actual array and specifies the size of the array. Notice the strange syntax: the type is followed by a number in parentheses, then by an empty set of braces, which are necessary to distinguish the right side of the equation from:

```
New Integer(5)
```

which simply declares a single integer, initializing it to 5.

The braces also serve as an initializer. Since the braces in the previous code have nothing inside of them, no initialization is taking place. Here, then, is an example of an array with initialization:

```
Dim myarray() As Integer  
myarray = New Integer(5) {0, 1, 2, 3, 4, 5}
```

Or you can initialize a multidimensional array, like so:

```
Sub ArrayTest()  
    Dim myarray(,) As Integer  
    myarray = New Integer(3, 1) {{0, 0}, {1, 2}, {3, 4}, {5, 6}}  
    MsgBox(myarray.Length)  
End Sub
```

First, notice how I declared the `myarray` variable in the first line: I put a lone comma inside the parentheses. This tells the compiler that I'm declaring a two-dimensional array variable, but not yet associating it with an actual array object. Then in the second line, I create the array object. It is a 4-by-2 array since the first dimension runs from 0 to 3, and the second dimension runs from 0 to 1. Then I initialize the elements by putting braces of arrays inside the outer braces.

Finally, as an experiment, I show the length inside parentheses. When I ran this, I saw the number 8. That's the total number of integers inside the array.

Strings in VB.NET

If you are a C or C++ programmer, you will be very happy to see the strong support of strings that is built into VB.NET. The `String` type supports both a plus (+) operator and an ampersand (&) operator, both for concatenation, with slightly different features. When you are combining strings, they both work the same way. Here's an example:

```
Sub StringSamples()  
    Dim mystr As String = "hello"  
    mystr = mystr + " there "  
    MsgBox(mystr)  
End Sub
```

When you run this, you will see the phrase "hello there" appear in the message box. Alternatively, you can use the & operator:

```
mystr = mystr & " there "
```

The difference between the + and & operators is apparent when you're combining a string with a variable of another type. The & operator converts all items to a string, as in this sample:

```
Sub StringSamples()  
    Dim mystr As String = "Number "  
    Dim num As Integer = 100  
    mystr = mystr & num  
    MsgBox(mystr)  
End Sub
```

When you run this, you will see the string "Number 100" appear in the message box.

The + operator, in contrast, converts strings to long integers. Thus, other than strings, you can only use the + operator with any of the number types. Here's an example:

```
Sub StringSamples()
    Dim mystr As String = "50"
    Dim num As Integer = 100
    Dim total As Long = mystr + num
    MsgBox(total)
End Sub
```

When you run this, you will see the number 150 appear in the message box. That's because the third line, which adds `mystr + num`, converts the string "50" to a number 50, and then adds it to the number 100 to get 150.



If you need to append to an existing string, instead of typing, for example, `mystr = mystr & "Hello"`, you can instead type `mystr &= "Hello"`. This also works with the + operator, as in `mystr += "Hello"`. If you're a C++ programmer, you will be pleased to see the += and &= additions to the Visual Basic language.

Copying strings couldn't be easier. Just set one string variable equal to another:

```
Sub StringSamples()
    Dim mystr1 As String = "George"
    Dim mystr2 As String
    mystr2 = mystr1
    MsgBox(mystr2)
End Sub
```

You can also compare strings using a single equal sign. (For the C, C++, C#, and Java programmers, remember VB.NET uses only a single equal sign for comparisons.) Here's an example:

```
Sub StringSamples()
    Dim mystr As String = "abc"
    Dim mystr2 As String = "abc"
    If mystr = mystr2 Then
        MsgBox("The strings are equal")
    End If
End Sub
```

(As for the syntax of the if statement, I talk about control structures in the "VB.NET Control Structures" section later in this chapter.)

Since String variables are actually instances of the .NET class String, you have access to all the String class member functions. For example, the following function converts a string to uppercase:

```
Sub StringSamples()
    Dim mystr As String = "George"
```



```

    mystr = mystr.ToUpper()
    MsgBox(mystr)
End Sub

```



The .NET strings have a very important feature: they are *immutable*; that is, you can't change them. At first that sounds like the most horrible thing a developer could do when creating a string class. But it's not as bad as it sounds: You can't modify the string, but you can put an entirely new string inside a string variable. (To the C and C++ programmers, what you're really doing is pointing the variable to a new String instance.) Thus, `mystr.ToUpper()` doesn't modify `mystr`; rather, it just returns a new string that's an uppercase version of `mystr`. If you want to store this back in `mystr`, you simply type:

```
mystr = mystr.ToUpper()
```

Therefore, don't fall into the trap of expecting your member functions to modify the string. They won't: They return a new string that you need to store back into the original variable. (If you have many long strings and are going to be modifying them, you might take a look at the `StringBuilder` class, which is documented in the online help. It's a true mutable string class.)

Some of the `String` member functions are members of the class itself, not of the instances. (For C and C++ programmers, I'm talking about static members here.) To access these methods, simply type the word `String`, then a period, and then the function name. Here's an example:

```

Sub StringSamples()
    Dim mystr As String = "George"
    Dim mystr2 As String = "GEORGE"
    If String.Compare(mystr, mystr2, True) = 0 Then
        MsgBox("The strings are equal, ignoring case")
    End If
End Sub

```

The function call `String.Compare(mystr, mystr2, True)` performs a comparison, while ignoring case. Thus, "George" and "GEORGE" will show as being equal by the `Compare` function. (If you leave off the `True`, or use `False` instead, you'll do a case-sensitive comparison.) The `Compare` function returns a value of 0 if the strings are equal.



Strings in .NET are zero-based, meaning each character in the string has an index; the first gets index 0, the second character in the string gets index 1, and so on. This is unlike previous versions of Visual Basic, where the index of the first character in a string was 1.

Here's a short list of some of the more useful string member functions. (For the complete list, check out the Visual Studio .NET online help's index under "String Class.")

All of these functions apply to an instance of `String`, unlike the `Compare` function I described earlier, which is a member of the `String` class itself.

Chars. Technically, `Chars` is a property, but it works like a function. Use `Chars` to retrieve individual characters in a string. For example, if your string variable called `mystr` contains “Hello”, then `mystr.Chars(1)` will return the character “e,” while `mystr.Chars(0)` will return the first character, “H.”

EndsWith. Use this function to determine if the string ends with a substring that you specify. For example, if your string variable `mystr` contains “Hello there”, then `mystr.EndsWith("There")` will return `True`.

IndexOf. This function finds the first occurrence of a substring that you specify. For example, if your string variable `mystr` contains “This is a test, isn’t it?” then `mystr.IndexOf("is")` will return 2.

Insert. Use this function to insert a string into another string. However, remember you aren’t really changing the string; instead, you’re creating a new string equal to the original with the new string inserted. For example, if your string variable `mystr` contains “Hellome”, then `mystr.Insert(5, " from ")` will return a new string containing “Hello from me”.

PadLeft. Use this function to add spaces (or any character) to the beginning of a string (that is, the left side of the string) to make the string a certain length. For example, if your string variable `mystr` contains “computer”, which is eight characters, then `mystr.PadLeft(12)` will return the string “ computer”. You can also add any character you want, not just spaces, by specifying the character as the second parameter. Then, `mystr.PadLeft(12, " *")` will return the string “****computer”..

PadRight. This function is identical to `PadLeft`, except it pads on the right.

Remove. This function removes characters from a string. If `mystr` is “George Washington”, then `mystr.Remove(1, 5)` will return “G Washington”.

Replace. This function replaces a substring in your string with another substring, both of which you specify. For example, if `mystr` is “George Washington of Washington”, then `mystr.Replace("Wash", "Dry")` will return “George Dryington of Dryington”. Notice that it replaces every instance of the first given substring.

Split. Use this function to split a single string into an array of strings. The original string gets split up based on a character you specify. For example, if `mystr` is “Once upon a time.”, then `mystr.Split(" ")` will return an array containing “Once”, “upon”, “a”, and “time.”. Here’s a sample macro that splits a string and displays each string of the resulting array in a message box:

```
Sub StringSamples()
    Dim mystr As String = "Once upon a time."
    Dim myarray As String() = mystr.Split(" ")
    Dim onestr As String
    For Each onestr In myarray
        MsgBox(onestr)
    Next
End Sub
```

ToLower. This function converts the string to lowercase, returning the new string consisting of all lowercase letters.

ToUpper. This function converts the string to uppercase, returning the new string consisting of all uppercase letters.

VB.NET Control Structures

VB.NET supports all the usual control structures that you would expect in a modern programming language. In the sections that follow, I discuss conditional statements, loop statements, and exceptions.

Conditional Statements

VB.NET supports both `If` statements and `Select/Case` statements. (For C++ programmers, `Select/Case` statements are the same as `switch` statements.)

There are two forms of the `If` statement: a standard `If` statement and a block-`If` statement. Here's an example of a standard `If` statement:

```
If size < 50 Then MsgBox("Size is less than 50")
```

This statement occupies only a single line. It always starts with the `If` keyword, then a condition, the word `Then`, and finally the statement to execute, if the condition is true.

Now here's an example of a block-`If` statement:

```
If size < 50 Then
    MsgBox("Size is less than 50")
    size = 50
    MsgBox(size)
End If
```

The key here is that the first line contains the keyword `If`, the condition, and the keyword `Then`. The lines that follow, up to the `End If` line, only run if the condition is `True`. The `End If` line signifies the end of the `If`-block.

The block-`If` statement can also support multiple `Else` blocks:

```
Sub Conditionals()
    Dim size As Integer = 125
    If size < 50 Then
        MsgBox("Size is less than 50")
        size = 50
        MsgBox(size)
    ElseIf size < 100 Then
        MsgBox("Size is less than 100")
    Else
        MsgBox("Size exceeds 100")
    End If
End Sub
```

Notice the keywords used: the `ElseIf` keyword is a single word, and the final block simply uses an `Else` statement.

VB.NET also supports a `Select/Case` statement. Here's an example:

```
Sub Conditionals()  
    Dim size As Integer = 5  
    Select Case size  
        Case 0  
            MsgBox("It's 0")  
        Case 1, 3, 5, 7, 9  
            MsgBox("Odd, less than 10")  
        Case 2, 4, 6, 8  
            MsgBox("Even, less than 10")  
        Case 10 To 20  
            MsgBox("Between 10 and 20")  
        Case Else  
            MsgBox("Greater than 20 or less than 0")  
    End Select  
End Sub
```

The first line is the variable being tested, in this case, `size`. The case lines that follow are the items against which to compare the variable. You can either put a single value or a list of values separated by commas; or, if you're dealing with integral types, a lower bound and an upper bound separated by the word `To`. Optionally, you can have a final `Else` clause that runs for all other cases.

Unlike C and C++, you can put strings in a `Case` statement. Thus, the following is perfectly valid:

```
Sub Conditionals()  
    Dim name As String = "Fred"  
    Select Case name  
        Case "Fred", "Ethel"  
            MsgBox("It's the neighbors")  
        Case "Lucy", "Ricky"  
            MsgBox("Its us")  
    End Select  
End Sub
```

Loop Statements

There are four loop statements that you can use in VB.NET: a `While` loop, a `Do` loop, and two forms of a `For` loop.

First, here's an example of the `While` loop:

```
Sub Loops()  
    Dim n As Integer = 0  
    While n < 5  
        MsgBox(n)  
        n += 1  
    End While  
End Sub
```

```
End While
End Sub
```

You can see that the `While` loop starts with the keyword `While`, then a condition. The lines that follow are those the `While` loop executes while the condition is `True`. You signify the end of the lines with the `End While` statement. Also, remember that if the condition is not satisfied initially, the lines in the `While` loop will not execute at all, and execution will resume after the `End While` statement.

Now here's an example of a `Do` loop:

```
Sub Loops()
    Dim n As Integer = 0
    Do
        MsgBox(n)
        n += 1
    Loop While n < 5
End Sub
```

This works the same as the `While` loop, except the computer tests the condition after the lines inside the block run. Therefore, the lines inside the block always run at least once, even if the condition is not satisfied.

You can also code the `While` loop using syntax from the `Do` loop. The following loop performs identically to the first `While` loop in this section; the syntax is just slightly different:

```
Sub Loops()
    Dim n As Integer = 0
    Do While n < 5
        MsgBox(n)
        n += 1
    Loop
End Sub
```

And now here's an example of the first kind of `For` loop. This kind is for simple counting:

```
Sub Loops()
    Dim n As Integer
    Dim a As Integer = 1
    For n = 1 To 15
        a += n
    Next
    MsgBox(a)
End Sub
```

The `For` loop starts with the keyword `For`, then the variable that serves as the counter, then the starting number for the counter, the word `To`, and the ending number for the counter. In this example, the `For` loop will run 15 times. The first time it runs, `n` will be 1. The final time it runs, `n` will be 15.

Optionally, you can tell how many times to increase the counter variable after each iteration. By default, the counter variable increases by 1. The following code, however, increases it by three each time by appending `Step 3` after the `For` loop's header line:

```
Sub Loops()  
    Dim n As Integer  
    Dim a As Integer = 1  
    For n = 1 To 16 Step 3  
        a += n  
    Next  
    MsgBox(a)  
End Sub
```

To count down, simply use a negative number for the `Step`, as in:

```
For n = 16 To 1 Step -3  
    a += n  
Next
```

Finally, when dealing with container classes, you can iterate through the containers using the `For Each` construct. One common place to do this is with arrays. Here's an example that iterates through the items in an array:

```
Sub Loops()  
    Dim myarray() As Integer  
    myarray = New Integer(5) {0, 1, 2, 3, 4, 5}  
    Dim n As Integer  
    For Each n In myarray  
        MsgBox(n)  
    Next  
End Sub
```

Exceptions

Exceptions are an important addition to the VB.NET language. They work similarly to other languages, such as C++.

There are three subblocks in an exception block. These are:

Try block. This is the block that contains the code that might result in an exception.

Catch block. This block contains the exception handler. You can have additional `Catch` blocks to handle more exceptions.

Finally block. This block contains the code that always runs, whether an exception occurred or not.

Here's an example of an exception handler dealing with file I/O. File I/O is probably the most error-prone situation, since pretty much anything unexpected can

happen: the disk could fill up; another program could corrupt a file that your program is working on; and so on. This example opens a file, writes several lines of text to it, and closes it.

```
Sub WriteFile()  
    Try  
        Dim f As TextWriter  
        Dim n As Integer  
        f = File.CreateText("c:\myfile.txt")  
        For n = 1 To 100  
            f.WriteLine("Number " & n)  
        Next  
        f.Close()  
    Catch e As IOException  
        MsgBox("Exception occurred " & e.Message)  
    Finally  
        MsgBox("All finished!")  
    End Try  
End Sub
```

If you're trying this out, please refer to "Other Ways to Customize Visual Studio .NET" in Chapter 1, "All about Macros and Add-ins," for information on how to set up your macros. You will need to include the following line for this preceding code to work:

```
Imports System.IO
```



One problem in working with exceptions is determining which exceptions can occur. In the previous example, I knew that the most likely culprit to raise an exception would be the `WriteLine` method. To find the exceptions this method can raise, I opened the online help and located the `TextWriter` Class in the index. Then I chose the All Members item under the `TextWriter` Class entry. I located the `WriteLine` method and opened up the one that takes a string as a parameter. I then found, midway down the page, a list of the exceptions the `WriteLine` method can throw. There was only one, and it was `IOException`.

Classes in VB.NET

Classes with support for inheritance are a welcome addition to Visual Basic. Here's an example of a class in VB.NET:

```
Class DrawingShape  
  
    Protected width As Integer
```

```
Protected height As Integer

Function GetArea() As Integer
    If VerifyDims() Then
        Return CalculateArea()
    Else
        Return -1
    End If
End Function

Protected Overridable Function CalculateArea() As Integer
    Return 0
End Function

Private Function VerifyDims() As Boolean
    If width < 0 Or height < 0 Then
        VerifyDims = False
    Else
        VerifyDims = True
    End If
End Function

Sub New(ByVal x As Integer, ByVal y As Integer)
    width = x
    height = y
End Sub

End Class
```

In a moment I'll show you an example of two classes derived from this `DrawingShape` class. But, first, some points about it:

- Access levels go to the left of an item and refer only to the item that follows (unlike C++). Your choices for access levels are `public`, `protected`, or `private`. Additionally, VB.NET has a scope called `friend`, which means the items are private outside the module containing the class, but public anywhere within the module. If you do not specify an access level, an item will be public by default (except for constants, which default to private).
- To return data from a function, you can either use the `Return` statement (as I did in the `GetArea` function) or set the function name equal to the return value (as I did in the `VerifyDims` function). Note, however, that most people prefer to use the `Return` statement.
- To specify that derived classes can override a function, put the keyword `Overridable` after the access level. You can see this in the `CalculateArea` function.
- The `New` function is a constructor. The constructor gets called when you create an instance of the class. You can see this constructor takes two integer parameters.

Now here's an example of two classes derived from class `DrawingShape`:

```
Class Rectangle
    Inherits DrawingShape

    Protected Overrides Function CalculateArea() As Integer
        Return width * height
    End Function

    Sub New(ByVal x As Integer, ByVal y As Integer)
        MyBase.New(x, y)
    End Sub

End Class

Class Triangle
    Inherits DrawingShape

    Protected Overrides Function CalculateArea() As Integer
        Return width * height / 2
    End Function

    Sub New(ByVal x As Integer, ByVal y As Integer)
        MyBase.New(x, y)
    End Sub

End Class
```

Each of these classes is derived from the `DrawingShape` class. Notice the second line of each class, `Inherits DrawingShape`, which declares the base class.

Each of these classes also overrides the `CalculateArea` function. To override it, I included the `Overrides` keyword (not to be confused with the `Overridable` keyword I used in the `DrawingShape` class).

Finally, each class also has a constructor. In VB.NET, you do not label your constructors as `Overridable` or as `Overrides`. (If you're a C++ programmer, this shouldn't be a surprise to you. The same is true with the ANSI standard for C++. Under the ANSI standard, constructors are not virtual.) Also, when creating a constructor for a derived class, you must call the base class constructor in the first line of the derived constructor's code. You can see this line in constructor: `MyBase.New(x, y)`. (You always use the keyword `MyBase`.) Finally, in VB.NET, when you have a constructor in a base class that takes parameters, you must provide a constructor with the same parameter types in the derived class, even if the constructor in the derived class does nothing more than call the constructor for `MyBase`.

Other VB.NET Topics

In this section I provide you with some miscellaneous topics in VB.NET programming.

Comments. In VB.NET, a single quote character begins a comment. The comment runs to the end of a line. A comment can begin in the middle of a line. Here are some samples:

```
' Loop through each string in the array
For Each onestr In myarray
    MsgBox(onestr) REM display the string
Next
```

Breaking up lines. Unlike other languages where statements end with a semicolon, statements in VB.NET end with the end of a line. If you have a really long line, you can just put it on one line, scrolling off the right side of the code editor. But for esthetics, you might want to split up the line. To split a line in VB.NET, end the first line with a space and then an underscore; you're then free to continue the statement on the next line. Of course, you can only split a line in places where you would normally insert a whitespace; you can't, for instance, break a line in the middle of a variable name. And you're also free, by ending a line with an underscore, to stretch a statement over as many lines as you need.

Displaying output to the user. There are three ways you can display output:

- *Using the MsgBox function.* You can use this function to display a small message box showing a string you pass to the `MsgBox` function. Many of the examples in this chapter used the message box.
- *Writing to the status bar.* For occasional single-line messages, you can display a line of text in the main IDE's status bar. Here's a sample line of code that does this:

```
DTE.StatusBar.Text = "Finished processing."
```

- *Writing to an output pane.* For sophisticated output, you can create a new pane in the output window and write to it. Here are two helper functions to help you do this, followed by a sample macro that uses the two functions. First, the two functions:

```
Function AddOutputPane(ByVal title As String) As OutputWindowPane
    Dim outwin As Window = DTE.Windows.Item _
        (EnvDTE.Constants.vsWindowKindOutput)
    outwin.Visible = True
    Return outwin.Object.OutputWindowPanes.Add(title)
End Function
```

```
Sub Print(ByVal Output As OutputWindowPane, ByVal text As String)
    Output.OutputString(text & Chr(13))
End Sub
```

- The first of these shows the output window, and then adds a new pane to the output window. The function returns a reference to the new output pane, which you can pass to the second function. The second function takes as a parameter the output pane and a text string, and it writes the string to the pane, along with a carriage return at the end. And now here is a sample macro that uses these two functions:

```
Sub SetupOutputPane()  
    Dim MyOutput As OutputWindowPane  
    MyOutput = AddOutputPane("Macros")  
    Print(MyOutput, "This is one line")  
    Print(MyOutput, "This is another line")  
End Sub
```

Finally, another way to interact with the IDE user is by creating a window (called a form, in macro language), populating it with controls and interacting with the window. In other words, the way to interact is to build a full-blown GUI. I talk about this in Chapter 4, “Macros That Interact with the User.”

Moving Forward

This chapter provided you with a brief introduction to VB.NET. It gave you enough information to get you up to speed on the language that you use to write macros in Visual Studio .NET. That means I couldn't detail every aspect of the VB.NET language, as that would require an entire volume or more.

If you're interested in learning more about VB.NET as you write your macros, I suggest exploring the online help. It's easy to navigate, and you should be able to quickly find the answers to any of your questions. But if you're interested in mastering VB.NET as a full-scale programming language (beyond macros), then I recommend playing with the main VB.NET development tool within Visual Studio .NET and studying the online help. Additionally, you might pick up a book or two about VB.NET. There are plenty out there, and many of them are excellent.

In the next chapter I detail the process of building a macro and interacting with the Visual Studio .NET IDE.

Introducing the Visual Studio Macro IDE

In this chapter I show you how to get going quickly with macros. First I talk about the different parts of the macro projects and how they fit together. Next I take you through the Macros IDE, showing you all the different ways you can manage multiple projects at once. After that, I describe how the main IDE provides some shortcuts for quickly entering a macro using what is called a *temporary macro*. Temporary macros are a convenient way for quickly automating simple tasks in the IDE. Moreover, as you'll learn, these macros need not remain temporary: When you create one of these, you can rename it, which will make it permanent, and then modify and improve it as you create a more powerful, complete macro.

The Macro System and the Main IDE

Macros live in projects, which are much like the programming projects you're already familiar with. But two points are very important in understanding how the macro system works together with the main IDE:

- *Macro projects and programming projects are separate.* When you write a macro, it is not a part of a programming project or solution. Instead, it stands on its own, and you can load it at any time, regardless of the programming project or solution you are presently working on. If you have a macro project open, to which you have made changes that you have not yet saved, and you close the current solution you are also working on, you will *not* be prompted to save the current

macro project. Instead, the solution will close, but the macro project will remain open. Only when you unload a macro project or shut down Visual Studio .NET will the main IDE ask if you want to save the changes to the macro project.

- You choose which macro projects you want to have open. You are free to have any one or multiple macro projects open simultaneously. Then when you restart the Visual Studio .NET, the macro projects load when Visual Studio .NET loads. Thus, you can have macros that respond to the launch of Visual Studio .NET.

Macro development takes place from two sides: from the main IDE and from the Macros IDE, which is shown in Figure 3.1. When you work on macros from the main IDE, you interact with the macros through the Macro Explorer. However, even if you plan to develop your macros using the main IDE (which you will be doing if you plan to do any macro coding), I caution you not skip the section in this chapter called “Working with the Macro Explorer,” for there I cover many important aspects of macro development that are applicable whether you use the Macro Explorer or not.

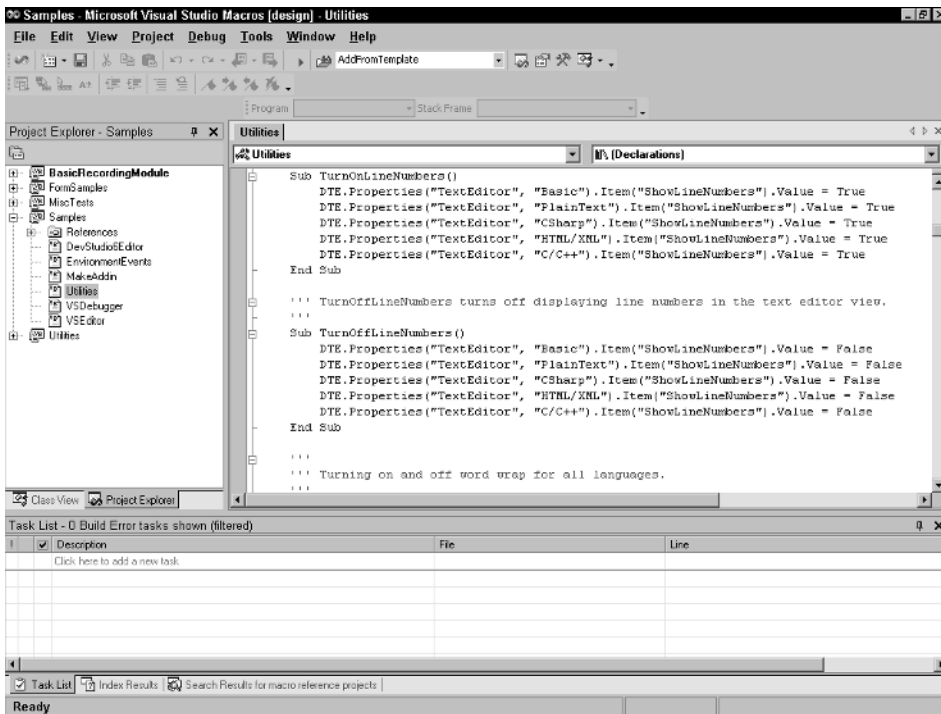


Figure 3.1 The Macros IDE.

Macro Development Divisions

The Visual Studio .NET macro system is divided into several parts that help you manage your macros. At the highest level is the macro system itself. Think of the macro system as the equivalent of a solution in the main IDE: The macro system holds your macro projects in the same way that a solution holds your programming projects. However, unlike solutions, there's only one macro system.

Projects and Modules

The macro system includes separate macro projects that you create, save, load, and unload. A macro project is a single entity with its own variables, macros, functions, and classes. The macro project exists in a single file on your hard drive. This file gets a .vsmacros extension. Having a single file means that if you want to share a macro project with somebody else, you can easily give that person the macro project file, allowing the recipient to use the macros in the project.



If you double-click a .vsmacros file in Windows Explorer (outside of Visual Studio .NET), one of two things will open: First, if Visual Studio .NET is not running, it will start up and load the macros project. Second, if Visual Studio .NET is already running, it will load the macros project.

When you create multiple macros, think of a project as its own *program*, just as you build a project in the main IDE into a standalone program or library. A macro project has its own variables, which are not accessible to the subroutines and functions in other macro projects.

When you organize your macro projects on your hard drive, you can put each project in its own directory—but you don't have to. Since each project uses only a single file, you can have multiple .vsmacros project files in a single directory, with each file containing a separate project.



Be careful if you rename your macro files. If you change `MyMacros.vsmacros` to, for example, `SystemUtils.vsmacros`, you will be changing only the filename, not the project name. Then, when you open the macro project, you will see the previous name in the Macro Explorer, not the new name that you gave the project. Therefore, I recommend you rename your macros from the Macro Explorer, as I describe in the section "Working with the Macro Explorer."

Inside the macro project you create modules. Think of a module as a source code file, even though all the modules for a project are saved inside the single project .vsmacros file. You write your code inside a module. You can have multiple modules in a single macro project, or you can just have one. (If you have no modules, the macro project won't be of much value since it won't have any code.)

Technically, a module isn't quite analogous to a source code file. The reason is that even though the Macros IDE considers a module a standalone file that you edit, you can actually put two modules inside a single module file. However, I recommend that you keep each module in its own file, in the interest of keeping your macro project manageable.

Here's an example of an entire module:

```
Imports EnvDTE
Imports System.Diagnostics
Imports System.Windows.Forms

Public Module MainUtilities

    Sub ShowCurrentDate()
        MsgBox(Now)
    End Sub

    Sub ShowUsername()
        MsgBox(SystemInformation.UserName)
    End Sub

End Module
```

The first section contains the *imports*. Every class in the .NET framework exists inside a *namespace*, and to use the class you must do one of two things: *fully qualify* the class name by specifying the namespace, then a dot, then the class name; or you must provide an *imports* statement. Notice the third *imports* line imports the namespace `System.Windows.Forms`. The reason is that the `SystemInformation` class, which I use in the `ShowUsername` subroutine, is part of the `System.Windows.Forms` namespace. (`UserName` is a property of the `SystemInformation` class.) Thus, I had a choice:

- I could fully qualify the `SystemInformation` class, like so:


```
MsgBox(System.Windows.Forms.SystemInformation.UserName)
```
- I could list an *imports* statement at the top, as I did, and then have direct access to the class name `SystemInformation`.

Next is the module declaration. This consists of the words `Public Module`, followed by the name of the module. Each module name within a project must be unique. The module begins with the module header and ends with the statement `End Module`.

Finally, inside your module you put your individual macros. The macros are VB.NET subroutines. In addition you can have variables, classes, and functions, and other subroutines that are not macros. The extra source code provides support for your macro routines.



Macros are public subroutines that do not take any parameters. By nature of a subroutine, macros do not have a return value. (That's because VB.NET separates subroutines from functions: functions have a return value, subroutines do not.) Optionally, you can declare a subroutine as private, making it unavailable as a macro. See the "Scoping" subsection for information on public and private subroutines.

Class Files and Code Files

If you're familiar with VB.NET, you will be interested to know that you can also insert class files and code files into your project. While working inside the Macros IDE, you can choose File→Add New Item. The Add New Items dialog box will open, showing three templates: Module, Class, and Code File.

As with modules, the class file and code files are not really files; they're parts of your project that live inside the .vsmacros files.

Be careful when working with class files and code files, however, because these items are not accessible as macros. Instead, treat them as having supporting roles in your projects. Although you can write code for a module inside your code file, the module will still not be visible as containing macros. Therefore, subroutines inside the module cannot function as macros.

Scoping

Modules have an interesting feature in that all the variables and other identifiers you declare in your module are accessible throughout the macro project on a global level, which means you do not have to fully qualify the names. For example, suppose you have a project with the `MainUtilities` module shown earlier in the "Projects and Modules" section. Then suppose you add another module that looks like this:

```
Imports EnvDTE
Imports System.Diagnostics

Public Module TestModule

    Sub CallFriend()
        ShowUsername()
    End Sub

End Module
```

You can see that the `CallFriend` subroutine is calling a subroutine called `ShowUsername`. That subroutine is in the `MainUtilities` module. But can the `CallFriend` subroutine call a subroutine in another module without giving the module name? Yes, it can. You do not need to fully qualify it, although you can, if you want to make sure people reading the code will know where the `ShowUsername` function is (generally a good idea). You can fully qualify the call by replacing the `ShowUsername()` line with the following line:

```
MainUtilities.ShowUsername()
```

Even though the names in a module are globally available within a project, one somewhat strange feature is that you can have two modules with the same name in it. For example, you could put a subroutine called `GetInfo` in the `MainUtilities` module, and another subroutine also called `GetInfo` in the `TestModule` module. But, then, if you want to call one of the `GetInfo` subroutines, you must fully qualify the name as either `MainUtilities.GetInfo` or `TestModule.GetInfo`.

If there are any subroutines in a module that you don't want other modules to call, add the word `Private`, as in this subroutine:

```
Private InternalSize As Integer
Private Sub InternalData()
    InternalSize = 10
End Sub
```

You can see that I have two items in the preceding code: a private integer variable called `InternalSize` and a private subroutine called `InternalData`. To make each item private, I simply included the word “private” before the declaration to make private item unavailable to any items outside of the module: Other modules cannot access the private item, nor can the main IDE access the item.



Since the main IDE cannot access an item marked as private, that means if you have a subroutine that takes no parameters but you do not want it available as a macro, you can declare it as private.

Optionally, you can include the word `Public` before an item inside a module to make the item accessible from outside the module; however, that isn't necessary, because the items are public by default.

As for modules, you are free to name them as you please; however, you cannot have two modules with the same name in a single project.

Working with the Macros IDE

The Macros IDE makes developing macros easy. If you're reading this book straight through, then you've already learned a bit about the Macros IDE and seen a little of it in action. If not, you need to know that the Macros IDE is a complete IDE for developing macros that is separate from the main IDE. It looks very much like the main IDE, except its focus is on macros.



Remember, the macros IDE is not a standalone program. You cannot start it independently of the main IDE.

The Visual Studio .NET main IDE provides you with several ways to get to the Macros IDE:

- Double-click a module name in the Macro Explorer. Or right-click a module name and choose Edit. Either way, the Macros IDE will open, showing the code for the module.
- Right-click a macro name in the Macro Explorer and choose Edit. The Macros IDE will open, showing the module containing the macro, with the code editor's insertion point on the first line in the macro subroutine's code.

- From the main IDE, choose Tools→Macro→Macros IDE.
- Press Alt+F11.

When you are using the Macros IDE, I recommend opening its task list, shown in Figure 3.2 (choose View→Other Windows→Task List, or press Ctrl+Alt+K). Having the task list open is useful because as you type your VB.NET code, the Macros IDE continually checks the syntax of what you type. If the IDE finds errors in your code, it will display the errors in the task list. Therefore, you can constantly keep an eye on the task list to see if you've made any mistakes in your code.



Inside the Macros IDE, the Macro Explorer is not available. Instead, you have the Project Explorer, which serves the same purpose. But be aware that double-clicking a macro name in the Project Explorer opens the macro source code for editing; it *does not* run the macro.

The Macro Explorer in the main IDE and the Project Explorer in the Macros IDE are connected: If you make a change to the macro system using the Macro Explorer and then look at the Macros IDE's Project Explorer, you'll see the same changes. Similarly, if you make a change in the Macros IDE's Project Explorer and then switch to the main IDE, you'll see the change noted in the Macro Explorer.

The Parts of the Macros IDE

Like the main IDE, the Macros IDE has several tool windows that let you manipulate your macro projects. You can move these tool windows around in the same way you can in the main IDE. Here are the tool windows that you'll find useful:

Project Explorer. This window shows the projects that are currently loaded into the IDE. Under each project is a list of the files for the project. The Project Explorer does not list individual macros.

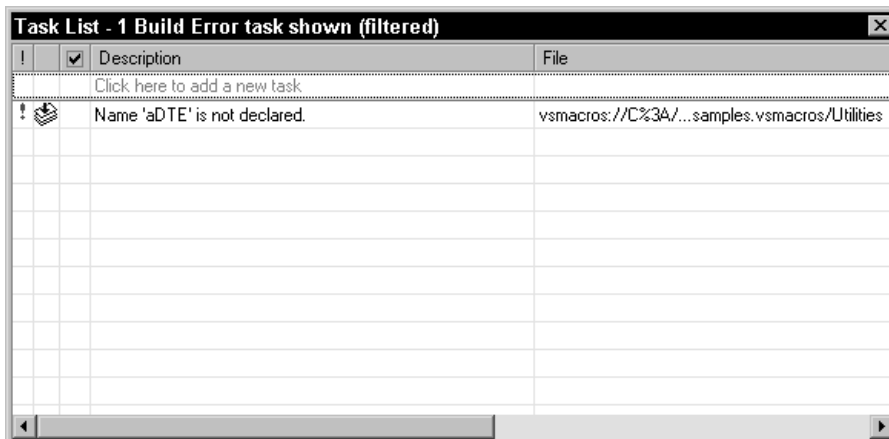


Figure 3.2 The Macros IDE task list.

Class Explorer. Although this window is named the Class Explorer, it shows your projects broken down by modules as well as classes. Further, under the modules, the Class Explorer includes the macro names that are part of the module. Normally you put modules and classes in their own files, and so typically the items in the Class Explorer will be the same as the items in the Project Explorer.

Help Index. This shows the index entries for the online help, as it does in the main IDE.

Help Contents. Like the same-named window in the main IDE, this window shows the contents of the help.

Dynamic Help. If you like this feature in the main IDE (a lot of people don't), it's available in the Macros IDE as well. It will track what you are doing and make suggestions for the appropriate help item.

In addition to the tool windows, there are document windows that hold your source code files. (But remember that these source code files aren't really individual files; they're treated as separate files by the IDE. Ultimately, though, all are saved together inside the single .vsmacros project file.)

The Macros IDE also has several useful toolbars. The list is smaller than that in the main IDE, but those that are present serve similar purposes as the like-named toolbars in the main IDE. They are:

Debug. Provides buttons for debugging a macro, such as tracing through the macro and stepping into its procedures.

Debug Location. Gives information about the currently running process and the call stack.

Full Screen. Includes a single button that lets you toggle a full-screen view of the source file you're currently editing.

Standard. Includes buttons for such activities as saving the project, searching for text, and quickly switching to the Project Explorer or the Class View.

Text Editor. Allows quick access for advanced text-editing features such as manipulating bookmarks in the code and indenting entire blocks of code.

You can choose which toolbars you want to show by right-clicking in the blank areas around the toolbars. The popup menu will list all the toolbars, with a check beside those that are showing.

Finally, like the main IDE, you can completely configure the Macros IDE. To do so, either choose Tools⇨Customize or right-click in the blank areas around a toolbar and choose Customize. The Customize dialog box will open, which works in precisely the same way as I described in Chapter 1, "All about Macros and Add-ins," in the section "Other Ways to Customize Visual Studio .NET": You can create and modify toolbars and manipulate the menus.

Managing Projects and Modules

If you're reading this chapter straight through, many of the topics in this section will be somewhat familiar later on when you read the section "Working with the Macro

Explorer.” The difference is that in the latter section I discuss how to manage your projects using the Macro Explorer in the main IDE; here I show you how to manage your project using the Macros IDE. However, there are certain things you can do only from within the main IDE through either the Macro Explorer or the menu items. These are:

- Create a new project.
- Load a project.
- Unload a project.

Since I show you later how to do these three tasks using the Macro Explorer, now I’ll show you how to do them using the *menus*. Remember, however, that these are the menus in the main IDE, not in the Macros IDE. (Why can’t life be less complicated?)



The Macros IDE is aware of an *active project*. You can tell which project is currently active by looking at the title bar of the IDE window. The title will start with the name of your project, followed by a hyphen, followed by the words “Microsoft Visual Studio Macros.”

Creating a New Project

To create a macro project, you must use the main IDE. To do so, choose Tools⇨Macros⇨New Macro Project. The New Macro Project dialog box will open. The only template available by default is Macro Project. Make sure the Macro Project template icon is highlighted. Then enter the name of your project in the text box labeled Name. Next, choose a directory in which to place your project.



When you create a new macro project, the Open dialog box creates a new directory for the project, just like the main IDE, which creates directories for you. The new directory has the same name as your project file, but without the .vsmacros extension.

After you create the new project, when you switch back to the Macros IDE, you will see the project in the Project Explorer.

Loading and Unloading Projects

Like creating a macro project, you can only load and unload projects from the main IDE.



Remember, the main IDE maintains a list of macro projects that are currently loaded. These are the macro projects that you (or your IDE users) can access while working with the IDE.

If there is a macro project that you don’t care to use or work on for the time being, you can remove it from the macro system. This is called *unloading the project*. After you do so, the macro project will vanish from the Macro Explorer in the main IDE and from

the Project Explorer and Class View in the Macros IDE. The project is still on your hard drive; it's just not loaded into the IDE.

Here's how to load or unload a macro project:

- *To load a macro project:* Choose Tools⇨Macros⇨Load Macro Project. A dialog box will open allowing you to locate and choose a .vsmacros project file. When you click Open, the macro project will be added to the macro system. It will also show up in the Macro Explorer. Alternatively, you can double-click the .vsmacros file in Windows Explorer.
- *To unload a macro project:* In the Macro Explorer, click on the name of the macro project you wish to unload. Then choose Tools⇨Macros⇨Unload Macro Project.

Saving Your Project

Back in the Macros IDE, to save a project, choose File⇨Save (the word "Save" on the menu will be followed by the name of your project). Or press Ctrl-S.

Which project you save depends on what item in the IDE is currently active:

- If you first click on any item in either the Project Explorer or the Class View, the project containing that item will be saved.
- If you first click on the source file window, then the project containing the active document will be saved.

You cannot tell the IDE to save only a single module. Since all modules within a project are stored in a single .vsmacros file, you can only save the entire project, including all its modules, at once.

Note also that when you edit a macro, you don't need to save the project to see the changes from within the Macros IDE. The main IDE knows of the code as it is in memory, not as it is on disk.

Although the macro system keeps all the modules for a project wrapped up inside a single .vsmacros file, you can export the individual modules to their own text files with a .vb extension. The macro system doesn't actually use these .vb files; the macro system still uses the modules as they are stored in the .vsmacros file. But this way you can use the files in other VB.NET projects if you wish. (To use the module in another macro project, you can use the insert the module by right-clicking on the macro project in the Project Explorer in the Macros IDE, and then choosing Add⇨Add Existing Item from the popup menu.)

To export a module to its own file, either make sure the module is open in the source code window or click on the module's name in the Project Explorer. Then choose File⇨Export, where the word "Export" will be followed by the name of the module you are exporting. An Export File dialog box will open that's equivalent to a typical Save As dialog box. Choose the location where you want to save the file, enter its name, and click Save.



When you export a file, the file will be saved in Unicode format. Therefore, some text editors might not be able to open it (although, believe it or not, Notepad.exe on Windows NT, 2000, and XP will open a Unicode file).

Default Macro Location

Although you can save your macro projects anywhere on the hard drive where you have permissions, by default, Visual Studio .NET stores the macros in a directory called VSMacros, in the default Visual Studio projects locations.

To view or change this default location, *make sure you're in the main IDE*, not the Macros IDE, and choose Tools⇒Options. Inside the Options dialog box, choose Environment⇒Projects and Solutions. In the options page is an edit control labeled "Visual Studio project locations." This edit control contains the default project path.



If you decide to change the default location for your macros to a different directory, remember that you will also be changing the default location for the programming projects and solutions in the main IDE.

Creating a New Module

To create a new module, you have some choices:

- Make sure the project that will contain the module is the active project. (You can tell by looking at the title bar of the Macros IDE (you should see the project's name there.) If your project is not the active one, click on its name (or one of its modules) in the Project Explorer. Next, choose File⇒Add New Item.
- Right-click the project name in the Project Explorer, and in the popup menu choose Add⇒Add Module.

Either of these two options will cause the Add New Item dialog box to open. Make sure Module is highlighted in the Templates list, then type the name of your new module in the Name edit box. Next, click Open. (Yes, the button is labeled Open, not Create or something more appropriate.)

After you click Open, the new module will be in your project. You will see it in the Project Explorer, and you can now edit it.

Renaming Projects and Modules

To rename a project or module from within the Macros IDE, do one of the following:

- Right-click the project name or module name in the Project Explorer and choose Rename. The project or module name in the Project Explorer will turn into an edit control, where you can type the new name and press Enter.
- Click the project name or module name in the Project Explorer, pause a couple seconds, and then click the name a second time. (Note, if the name is already highlighted, you should only have to click the name once.) The name will turn into an edit control into which you can type the new name.



Make sure that when you attempt to change a project's or a module's name that you are in the Project Explorer in the Macros IDE, not the Class View. You cannot change the project or module name from the Class View.

When you change a project's or a module's name, the change will not be permanent until you save the project by choosing File→Save, where the word Save will be followed by the project name. Even then, you will not see the actual .vsmacros filename change until you either unload the macro or shut down the main IDE, because the main IDE keeps the .vsmacros file open as long as the macro project is open.



Be careful with module names. The Project Explorer in the Macros IDE lists the name of the file that contains the module (even though technically all the "files" for these modules are stored inside a single .vsmacros file). This filename does not have to match the name contained in the header line of the Module declaration in the source code file. But that's when things get confusing. I prefer to name my module and its filename the same, for simplicity. Plus, that helps me to think of the file and the module as a single entity. But if you do rename the module, only do so by right-clicking the module in the Project Explorer and choosing Rename. That will keep everything synchronized.

Deleting a Module

To delete a module, do one of the following. (Make sure you do these in the Project Explorer, not the Class View.)

- Click the name of the module in the Project Explorer. (If an edit control appears so that you can edit the name, just press Esc.) Then either press Del or choose Edit→Delete.
- Right-click the name of the module in the Project Explorer and choose Delete.

Whichever action you take, you will then see a message that says, "Module1 will be deleted permanently" (but instead of Module1 you will see the name of your module). In other words, you cannot undo the deletion of a module. If you are sure you want to delete the module, click Yes. (The reason the action cannot be undone is that all the modules for a single project are all stored in a single .vsmacros file for the project. When you click Yes, the IDE deletes the module directly from the .vsmacros file. Therefore, although the .vsmacros project file is still present, the module is not.)

Running a Macro

To run a macro without starting a debugging session, either switch to the main IDE and double-click the macro name in the Macro Explorer, or, from the Macros IDE, click the mouse anywhere inside the macro's code in the source code editor; then either press Ctrl-F5 or choose Debug→Start without Debugging.



If there are errors anywhere in a macro project, then neither the main IDE nor the Macros IDE will let you run any of the macros in the project. Make sure, then, that there are no errors in your project.

An important point to be aware of when you run your macros is that if you have global variables in your macro project, these variables will not change between executions of the macro. For example, suppose you have this variable and macro defined in a module:

```
Private TestNumber As Integer = 10
Sub IncTestNumber()
    TestNumber += 1
    MsgBox("Test Number is now " & TestNumber)
End Sub
```

If you then go over to the main IDE and double-click the IncTestNumber name in the Macro Explorer, you will see a message box with the number 11. Now, based on what I said a moment ago, see if you can answer this question: The second time you click IncTestNumber, will you see 11 (meaning the variables got reset in between execution of the macro) or 12 (meaning the variables are still active)? If you answered 12, you were right.

However, if you change any code inside the project containing the macro (whether or not you save the changes to disk), then Visual Studio .NET will restart the project, meaning your variables will be reset as well. Thus, if I modify the IncTestNumber macro, when I return to the main IDE and double-click the IncTestNumber name, I will once again see the number 11, *not* 13, in a message box.



If you run a macro, and while it is running attempt to run another macro, the second macro will get queued, and will not begin until immediately after the first macro finishes.

Stopping a Macro

If a macro you have running gets stuck, or you just want to end it, you can force the macro to stop. Suppose you have the following macro (or perhaps a macro that calls this subroutine):

```
Public Sub WaitForAMoment()
    MsgBox("Going to sleep for 10 seconds...")
    System.Threading.Thread.Sleep(10000)
    MsgBox("Awake again!")
End Sub
```

Here, the Sleep function causes the macro to pause; you are required to specify the time in milliseconds. (There are 1000 milliseconds in a second.) I passed 10000 to the function, which means the macro will pause for 10 seconds. If you run this macro, the Visual Studio .NET will seem to freeze up for 10 seconds.

If, instead, you want to abort the macro before the 10 seconds is up, switch to the task bar on the Windows desktop. There, you will see an animated icon of a cassette tape flipping around. Double-click this icon and the macro will stop. Or, you can right-click this icon, in which case you'll get a popup menu with a single choice: "Stop Visual Studio macros." Choose this menu item to stop the macro.

Using the Code Editor in the Macros IDE

By now you've certainly seen the code editor in the Macros IDE, and you've probably discovered that it works pretty much just like the code editor in the main IDE. In the introductory section of Chapter 2, I mentioned that the code editor will automatically format your VB.NET code for you. Here, now, are some more tips for getting the most from the editor.

Collapsible Code

You may have noticed that to the left of some of your lines of code is a small minus sign that looks similar to the minus symbol in a treeview control. This sign serves the same purpose as that in a treeview control: to collapse the code. For example, if you want to see only a subroutine's header and hide the subroutine's code, you can collapse the subroutine. For example, if this is your subroutine:

```
Sub ShowCurrentDate()  
    MsgBox(Now)  
End Sub
```

and you click the minus sign that shows up in the code editor to the left of the Sub line, the code will collapse into just this line:

```
Sub ShowCurrentDate()
```

To the left of this single line will now be a plus sign instead of a minus sign, which you can click to reexpand the code. This collapsible code functionality is also called the *outlining* feature of the code editor.

And while the code is collapsed, you'll see to the right of it a small white box with an ellipses (...) in it. If you momentarily hold the mouse pointer over this box, a small tooltip window will appear showing you the collapsed code.



If you want to collapse all the subroutines and functions in your code, choose Edit⇨Outlining⇨Collapse to Definitions.

If you don't like this outlining feature, you can turn it off by choosing Edit⇨Outlining⇨Stop Outlining. Later, if you decide you want it back on, choose Edit⇨Outlining⇨Start Automatic Outlining.

Working with Blocks of Code

If there are several lines of code that you want to comment out, highlight them and choose Edit→Advanced↔Comment Selection. To uncomment a commented block of code, choose Edit→Advanced↔Uncomment Selection.

If you have the automatic formatting feature turned off (through the Tools↔Options dialog box, on the Text Editor↔Basic↔VB Specific page, under the Pretty Listing (reformatting) of code selection), you can automatically format a section of code by highlighting the lines and choosing Edit→Advanced↔Format Selection.

You can also perform all the usual features that are present in the main IDE under the Edit→Advanced menu, such as making a selection all uppercase or all lowercase (if you have such a need).



If you have a block of code collapsed and you want to perform a block edit operation (such as commenting out the code) on the collapsed code, highlight the box with the ellipses in it before performing the block edit operation.

Debugging a Macro

In this section I assume you're familiar with debugging concepts and how to use the debugger in the main IDE, which is the tool you use for debugging your programming projects. The debugger in the Macros IDE works very much like the one in the main IDE. However, there's a trick to getting your macros to run there:

1. To debug a macro, make sure the module containing the macro is opened in the code editor.
2. Click on the first line of the macro, the header line containing the Sub declaration.
3. Press F9 to set a breakpoint (or right-click the line and choose Insert Breakpoint). You will see a red highlight appear on the line.
4. Press F5 to start the debugger (or choose Debug↔Start).

When the debugger starts, it will break at the first line in your macro, and the first line's highlight will change from red to yellow.



When you are running a macro in the Macro IDE Debugger, you cannot stop the macro by double-clicking the macro icon in the tray of the Windows desktop's task bar. Doing so will have no effect.

While you are at a breakpoint, you can then do any of the following:

- *Step into a subroutine or function:* Choose Debug↔Step Into, or press F11, or click the Step Into button on the Debug toolbar.
- *Step over a subroutine or function:* Choose Debug↔Step Over, or press F10, or click the Step Over button on the Debug toolbar.

- *Step out of a subroutine or function:* Choose Debug→Step Out, or press Shift+F11, or click the Step Out button on the Debug toolbar.
- *Modify an existing breakpoint:* Right-click a line with a breakpoint and choose Remove Breakpoint, or Disable Breakpoint, or Breakpoint Properties. If you choose Breakpoint Properties, the Breakpoint Properties window will open, allowing you to add conditions upon which to break.
- *Add a breakpoint:* Right-click a line and choose Add Breakpoint.
- *Add a watch:* Right-click an identifier in the source code window and choose Add Watch. The Watch window will open, showing the identifier along with other identifiers you are already watching.
- *Add a quick watch:* Right-click an identifier in the source code window and choose Quick Watch. The Quick Watch window will open, showing you the identifier and its current value, which you can change.



When you're viewing the Quick Watch window, you can call other subroutines and functions in your code. Simply type the name of the subroutine or function in the Expression window along with parentheses containing zero or more parameters, and click Recalculate.

- *Watch the local variables in a subroutine:* Choose Debug→Windows→Locals.
- *Observe the call stack:* Choose Debug→Windows→Call Stack.

Working with the Macro Explorer

Although you will normally use the Macro IDE to develop your macros, you have full access to the macros from the standard IDE through the Macro Explorer. To view the Macro Explorer, shown in Figure 3.3, choose View→Other Windows→Macro Explorer.

Like the Solution Explorer, the Macro Explorer shows your macro projects in a hierarchical manner inside a treeview. When you expand a project, you see the individual modules, and under each module you see the individual macros for that project. (Remember, each macro is a public VB.NET subroutine inside the module.)

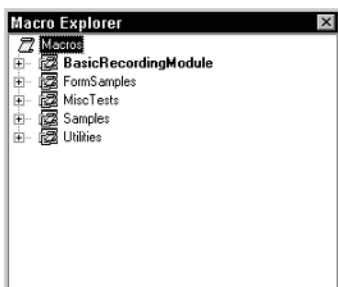


Figure 3.3 The Macro Explorer in the main IDE.



To run a macro using the Macro Explorer, double-click its name in the Macro Explorer or right-click its name and choose Run.

The Macro Explorer allows you to do the following actions on your projects, modules, and macros. First, here are items involving macro projects:

- *Create a new macro project:* Right-click the top item called Macros in the tree-view; in the popup menu, choose New Macro Project. The New Macro Project dialog box will open, just as it does when you use the menu items to create a new macro project. You only have one option for a template, Macro Project. Type a name and a location for your new project. Then click Open (even though you're actually *creating* and then opening the project).
- *Insert an existing project into the Macro Explorer:* If you have a macro project, such as one you received from somebody else, that's not in the Macro Explorer, you can add it. Right-click the Macros item and choose Add⇨Insert Existing Project.
- *To unload a project:* Right-click the project name and choose Unload Macro Project.
- *To save a project:* The only way to save a macro project from within the main IDE is to choose File⇨Save All. However, this saves everything that has changed. If you prefer to save only your file, use the Macros IDE and choose File⇨Save, where Save is followed by the name of your project.
- *To rename a project:* Right-click the name of the project and choose Rename.

Now here are items involving modules within a project:

- *To create a new module:* Right-click the module name in the Macro Explorer and choose New Module. The Add Module dialog box will open, allowing you to create the new module.
- *To rename a module:* Right-click the module name in the Macro Explorer and choose Rename. You can then type in a new name for the module.
- *To delete a module:* Right-click the module name and choose Delete. As in the Project Explorer in the Macros IDE, you cannot undelete a module once you've deleted it here. But you will see a slightly different message from the one you see in the Macros IDE: "Are you sure that you want to delete Module1? This action cannot be undone."
- *To Edit a module:* Double-click the module name. The Macros IDE will open and you will see the code file for the module whose name you double-clicked.

If you're digging through the popup menus as you read this, you'll see the menu name Set as Recording Project. For more information on this item, see "Quickly Recording a Temporary Macro" in this chapter.

And now here are the actions you can perform in the Macro Explorer on individual macros. Remember, macros are subroutines inside a module, and all the modules within a project are stored in a single .vsmacro file:

- *Create a new macro:* Right-click on the module that you want to hold the macro and choose New Macro. When you do so, the Macro IDE will open and the

Macro IDE will automatically insert a new macro called Macro1 (or Macro2, and so on) into the module.

- *Rename a macro:* Right-click on the macro name (back inside the Macro Explorer, in case you're looking at the Macros IDE now from the previous bullet item). Then choose Rename. You can type in a new name for the macro.
- *Delete a macro:* Right-click on the macro name and choose Delete. The IDE will ask if you really want to delete the macro, since the action cannot be undone. Just like modules, macros live inside a module, which lives inside the .vsmacros project file. If you delete a macro, the IDE completely removes it from within the .vsmacros project file; thus it cannot be undone. So be sure you really want to delete a macro before doing so.
- *Run a macro:* Either double-click a macro name or right-click on the macro name and choose Run.



While a macro is running, you will see two visual indicators: First, down in the status bar of the main IDE, you will see a small animated icon that looks sort of like a cassette tape flipping around; second, you will see this same animated icon in the tray portion of the taskbar on your Windows desktop.

Finally, remember this important tip:



When you double-click a macro in the Macro Explorer, the macro runs. Do not double-click the macro to edit it. However, if you double-click a module name, you will enter the Macros IDE so you can edit the module.

Quickly Recording a Temporary Macro

When programmers first developed the concept of a macro, the process was simple: The user identified a task that he or she wanted to repeat several times, and so started a macro *recorder*; next, the user performed the task to be repeated (such as typing something into the keyboard and perhaps selecting various menu items); finally, the user turned off the recorder. The macro was recorded. When the user wanted to repeat the task, he or she started the macro, usually by pressing some key.

The process was nice and simple. Fortunately, today, Microsoft has maintained that simplicity for those users who only want to record repeated tasks. The Visual Studio .NET IDE includes a recorder feature that's as easy as the steps just described. And it not only records the macros, but it saves the macros in the same language you use to write your own macros from scratch, VB.NET.

Saving the macros in VB.NET has a nice bonus feature, too: If you, the macro developer, want to automate a task that involves pressing keys, choosing menu items, and filling in dialog boxes, you can use the recorder features as a *starting point* for writing your macro.

Thus, in this section I show you how to get the most out of the recorder features.

Selecting the Recording Project

If you have no macro projects open and you start recording a macro, Visual Studio .NET will create a macro project for you by default. This project will be called `MyMacros`, and it will have two modules: one called `Module1` (which is a default module that the recorder doesn't use, but to which you're free to add macros) and another called `RecordingModule`. The macro that you record goes inside the `RecordingModule` module, and the macro is called `TemporaryMacro`.

But if you do have several macro projects open, you have a choice: You can either let Visual Studio .NET assign you a `RecordingModule` and a `TemporaryMacro` when it records your macro for you or you can choose which project will be the recording project. (If you let Visual Studio .NET assign you a `RecordingModule`, it will use the default `Module1` until you choose a different project to be the recording project.)



To choose the recording project, open the Macro Explorer, right-click the project, and choose Set as Recording Project. After you select which project will be the recording project, your chosen project's name will appear in boldface in the Macro Explorer.

Recording a Macro

Once you have chosen a recording project (which means you might have chosen the defaults), you can go ahead and record your macro. Now remember, the IDE will assign this macro the name `TemporaryMacro`, but it is in no way temporary: You are free to rename it, move it to another module, and so on.

If you are going to record a macro, I recommend opening up the Macro Explorer so that you can easily access the macro after you have recorded it. Then you're ready to record:



To record a macro, choose Tools → Macros → Record Temporary Macro. (The default key combination for this is Ctrl+Shift+R.)

When you begin recording, the Recorder toolbar will open. This toolbar lets you pause the recording, stop the recording, or altogether cancel the recording.

A point of note about the Recorder toolbar: Though it is a toolbar, it's special in that the IDE won't let you open it unless the recorder is running, nor can you close it when the recorder is running. Oddly, the IDE will, however, let you modify the toolbar while it is open (which means a macro is being recorded). But if you do modify the Recorder toolbar, your changes will not get recorded into the macro. (Probably, it's best not to mess with such self-referential universes. Leave that for the science fiction movies.)

While the Recorder toolbar is open, any keystrokes you perform, or toolbar buttons you click, or menu items you choose will be recorded. Further, if any of these actions opens a dialog box, and you click OK in the dialog box, your settings will be recorded. Each action that you perform will result in additional VB.NET lines of code being put in the macro subroutine.

The toolbar has three buttons on it:

Pause Recording. If you click this button, the recording action will be momentarily suspended, at which point any keystrokes or other actions you perform will not be recorded. Recording will resume when you click the Pause Recording button a second time.

Stop Recording. This button will end your recording and save the actions into the Module as a new macro.

Cancel Recording. This button is the abort button; if you click it, the recorder will stop recording and your actions will not be saved to a macro. Click this if you decide you don't want to record a macro after all.

Editing the Temporary Macro

After you have recorded a macro, you can edit it. To do so, locate the macro project in the Macro Explorer that is the recording project (its name will be bold). Underneath it, right-click on RecordingModule and choose Edit. When you do so, the Macros IDE will open.

Here's an example of a macro that I recorded. First, here are the actions that I recorded:

1. Ctrl+A (That's the same as Edit→Select All.)
2. Edit→Advanced→Format Selection
3. Edit→Copy
4. File→New File (In the New File dialog box, I chose Text File.)
5. Edit→Paste
6. File→Save As (In the Save As dialog box, I typed the same name as the file I copied the text from, but instead of a .cpp extension I used a .txt extension.)

And now here's the code I see in the Macros IDE for this module:

```
Option Strict Off
Option Explicit Off
Imports EnvDTE
Imports System.Diagnostics

Public Module RecordingModule

    Sub TemporaryMacro()
        DTE.Windows.Item("MyProgram.cpp").Activate()
        DTE.ActiveDocument.Selection.SelectAll()
        DTE.ExecuteCommand("Edit.FormatSelection")
        DTE.ActiveDocument.Selection.Copy()
        DTE.ItemOperations.NewFile("General\Text File")
        DTE.ActiveDocument.Selection.Paste()
        DTE.ActiveDocument.Save("C:\MyProject\ArrayTest.backup.txt")
    End Sub
End Module
```

```
End Sub
End Module
```

Notice that the Save selected the filename I typed in. That's not good; my intention was to write an automatic backup macro. But that's okay; I can change it. Also, notice that even though I typed `MyProgram.backup`, the macro recorded it as `MyProgram.backup.txt` with a `.txt` extension. But that's not the recorder's fault. That's the fault of the Save File As dialog box: If while inside the Save File As dialog box I had chosen All Files (*.*) for the Save as Type box, then Visual Studio .NET would have saved the filename as I requested it.

The next step is to modify this macro so it's more useful. I certainly don't want to save every file as `MyProgram.backup`. Instead, I will ask the IDE for the name of the document that's open, and I'll piece the filename together myself. Also notice that the first line in the macro is a call to `Activate`. That function activates the document called `MyProgram.cpp`. But in the final version of this macro, I hope to have the macro use whichever file is currently active. So I will chop that `Activate` line altogether.

To figure out the filename information, I'm going to use two .NET macro features:

- To retrieve the filename, I'm going to use the `DTE.ActiveDocument` object. That's a handy property right on the `DTE` object itself that returns an instance of a class called `Document`.
- To figure out the backup filename, I'm going to use a class called `Path`, which has a handy function called `ChangeExtension`. This function takes a filename and a new extension name as parameters, and returns a new string with the path modified. What could be simpler?

Finally, after I recorded the macro I decided that I should probably close the new backup file that I created. So I added a line at the end of the macro to close the file.

Here, then, is the new code after I changed it:

```
Option Strict Off
Option Explicit Off
Imports EnvDTE
Imports System.Diagnostics
Imports System.IO

Public Module RecordingModule

    Sub TemporaryMacro()
        Dim Filename As String
        Dim Doc As Document
        Doc = DTE.ActiveDocument
        Filename = Doc.FullName
        BackupFilename = Path.ChangeExtension(Filename, ".backup")
        DTE.ActiveDocument.Selection.SelectAll()
        DTE.ExecuteCommand("Edit.FormatSelection")
        DTE.ActiveDocument.Selection.Copy()
        DTE.ItemOperations.NewFile("General\Text File")
    End Sub
End Module
```



```
DTE.ActiveDocument.Selection.Paste()  
DTE.ActiveDocument.Save(BackupFilename)  
DTE.ActiveDocument.Close(vsSaveChanges.vsSaveChangesYes)  
End Sub  
End Module
```

Now this code is more usable in a general sense. Instead of working on a particular file, it works on any open file in the main IDE. Once I'm happy with the code, I can rename the macro and copy it to a different module from the RecordingModule. Then the macro is ready for prime-time use.

Assigning Shortcut Keys to Your Macros

If you give your macros to other people to use on their computers, you might not want them to have to keep the Macro Explorer open so they can access your macros. Fortunately, you have some choices on how to allow them to run your macros. (You can implement these on your own computer, too, for use during the development of your macros.)

- Choose and assign the macro a shortcut key so that when the IDE user presses a certain keystroke combination (such as Ctrl+Shift+m), the macro will run.
- Add various menu items to the user's IDE that allow menu access to the macros.
- Add various buttons to the toolbars on the user's IDE that allow button access to the macros.
- Provide instructions on how the user can choose to do any of the previous three items.

In Chapter 1, "All About Macros and Add-ins," in the section "Managing the Toolbars and Commands," I showed you how to add toolbar buttons for your macros. In the same chapter, in the section "Customizing the Menus," I showed you how you can insert menu items for your macros. Now here's how you can assign shortcut keys.

1. From the main IDE, choose Tools⇨Options. Expand the Environment tree; under Environment, click Keyboard to see the Keyboard options. (Alternatively, choose Tools⇨Customize. In the Customize dialog box, under the Toolbars tab, click Keyboard. This will also get you to the Keyboard options page.)
2. In the middle of the dialog box you'll see a listbox with all the commands known to the IDE. Either scroll through this to find your macro or just type the macro name in the text box above the list labeled "Show commands containing." Do not press Enter; just wait a moment and the list will shorten to only those commands that have the text you entered.

3. Next click on your macro in the list, then click the mouse on the text box labeled “Press Shortcut Key(s).” You will see the name of your keystroke appear in the edit control. For example, if you press Ctrl+Shift+I, you will see “Ctrl+Shift+I” appear in the text box. You will also see a list of places where this shortcut key is already assigned. (Hint: It’s a good idea to look at this list to make sure you’re not clobbering a shortcut key that’s already in use and that you use occasionally. Once, while using Microsoft Excel during a slow time in my mental activity, I reassigned Ctrl+S and later wondered why, when I tried to save my file, one of my macros would run instead.)
4. When you’re happy with the key assignment, press Assign.

Moving Forward

This chapter introduced you to the Macros IDE, which, as you now realize, looks very much like the main IDE. That means if you know how to use the main IDE, you’ll be comfortable using the Macros IDE as well. Further, you learned how to use the Macro Explorer in the main IDE to create and modify the names of your macros.

In the next chapter I show you ways to write macros that have their own user interfaces. And while on that topic, I’ll cover the different ways that you can present information to the user.

Macros That Interact with the User

In this chapter I show you how you can interact with the user in various ways. I devote the first half of the chapter to showing how you can reference other libraries in your projects, because to make interaction easy, often you will want to work with external projects or libraries. For example, you might have a .NET *assembly* (which is a .NET form of a .DLL) that contains subroutines, functions, or classes that you want to use in your macros. To use these items, however, you must set up a reference to the assembly in your macro project. Or you might have a set of subroutines and functions you wrote in VB.NET using the Macros IDE that you want to use in other macros.

In this chapter I also show you how you can access these subroutines and functions from any macro project using two different approaches: importing the code into your project and creating a standalone assembly that you reference in your project.

From there I move on to show you how easy it is to create forms and populate them with controls, and how you can interact with the forms. I also show how you can make use of the common Windows dialogs.

Finally, I discuss the different events you can respond to that occur in the main Visual Studio .NET IDE, events such as a window opening or the main IDE starting up.

Referencing Assemblies and Macro Projects

In Visual Studio .NET, you can create DLLs called assemblies. An assembly is a DLL or EXE that contains managed Microsoft.NET code, along with information about the file.

When you build a managed application or class library in Visual Studio .NET, the final .EXE or .DLL you create is an assembly.

Since your macros use VB.NET, you have access to the rich set of features included in .NET programming, including the use of external assemblies. (As a user of Visual Studio .NET, you've probably encountered assemblies before, but if you're not sure what they're all about, refer to Chapter 5, "Just Enough .NET Architecture.")

Referencing External Assemblies

When you want to use objects and classes from an external assembly, you need to add a reference to the assembly from your VB.NET macro. The Macros IDE includes a dialog box for adding references; the Macros IDE also includes a list of the referenced assemblies in the Project Explorer.



To view the references list in the Project Explorer, expand a project; the first item you will see under the project name is References. When you expand References, you will see the names of referenced assemblies.

To add a reference to your project, you have two choices:

- Right-click the word References under the project name in the Project Explorer. A popup menu containing a single item opens: Add Reference. Choose it and its dialog box will open.
- Make sure your project is active (in the Project Explorer or Class View, click on any item in the project; the project name will appear in the title bar of the Macros IDE). Then choose Project → Add Reference. The Add Reference dialog box will open.



Whenever you add a reference using the Project → Add Reference menu item, take a quick look at the title bar of the Macros IDE to make sure the correct project is active. It's easy to accidentally add the reference to the wrong project!

The Add Reference dialog box is shown in Figure 4.1. It contains a list of assemblies registered in the .NET system. You can add your project references to one or more of these assemblies. The list of assemblies contains the name of the assembly, the version number, and the path to the assembly's .DLL file.

To use the Add Reference dialog box, click on the assembly's name in the list and then click the Select button; or just double-click the assembly's name; its name will then appear in the Selected Components list in the lower portion of the dialog box. (You can select multiple names by holding down the Shift and Ctrl keys and clicking multiple names.) Repeat this step to add more than one assembly.

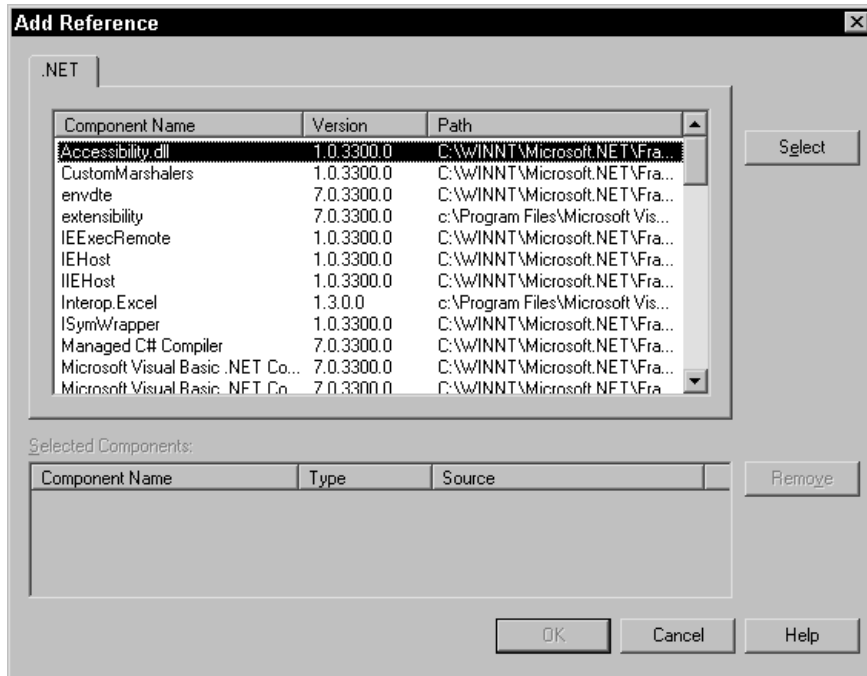


Figure 4.1 Use the Add Reference dialog box to add to your project references to external assemblies.

If you have a list of assemblies to add, but decide you want to remove one from the list, click its name in the Selected Components list and click Remove. When you're finished, click OK. (If you change your mind, you can always return to this dialog box later to add more references; or you can remove them from the References list in the Project Explorer. You will see that the References section in the Project Explorer now contains the items you chose in the Add Reference dialog box. If you want to remove a reference, right-click the reference name in the References list in the Project Explorer and choose Remove.

Referencing Items in Other Macro Projects

If you have a set of subroutines, functions, and classes in a macro project that you want to make available to other macro projects, you have a couple of choices for doing this. First, you can export the module containing the subroutines, functions, and classes, and then add the module to the project needing the items. Second, you can move the module into a VB.NET program inside the main IDE and build an assembly containing the items. (This second approach isn't as difficult as it sounds; in fact, it's pretty easy. The only catch is that you must have VB.NET available in your main Visual Studio .NET installation.)

LISTING REFERENCES FROM A MACRO

This is a book on macros, so in addition to telling you about references, I'm going to show you how you can gain access to a project's references at runtime. The following macro lists all the macro projects presently loaded, and for each project lists the references, including the full path to the reference. In the following listing, I obtain a Solution object from the `DTE.MacroSIDE` object. The Solution object contains a set of Project objects. I climb through the list of Project objects, and from each obtain a VSProject, which contains the list of references. For each reference, I ask for its name and its path, which I print to the Output window.

```
Imports EnvDTE
Imports System.Diagnostics

Public Module ObtainReferences

    Sub ListRefs()
        Dim a As VSLangProj.Reference
        Dim s As Solution
        Dim vsp As VSLangProj.VSProject
        Dim p As Project
        ClearOutput()
        s = DTE.MacroSIDE.Solution
        For Each p In s.Projects
            vsp = p.Object
            Print("Project: " & p.Name)
            For Each a In vsp.References
                Print("    " & a.Name)
                Print("        " & a.Path)
            Next
        Next
    End Sub

End Module
```

Both approaches have some pros and cons:

- *Exporting the module.* The advantage is that you have direct access to the source code in your macro project, because the module becomes an actual part of your project. The disadvantage is that your project ends up with a copy of the original module, rather than the original module itself. Thus, if you make changes to the original, you will have to duplicate the changes in your project, or reimport it.

- *Building an assembly.* The advantage is that when you modify the assembly, all the macro projects that reference the assembly pick up the changes. The disadvantage is that the code is no longer a macro in itself.

In this section I show you how to export a module and then import it to another project. In the next section, I show you how to transform your macro project into an assembly and then add a reference to the assembly.

To export a module:

1. Give the module a name that is going to be unique. Why? Because if you try to import the module to another project, and that project already has a module by the same name, the import will fail. Therefore, don't use the default name, `Module1`. (And when you change the name, change the module filename that shows up in the Project Explorer, as well as the name of the Module declaration inside the file. The best way to change both names simultaneously is by right-clicking on the module name in the Project Explorer and choosing Rename.)
2. In the Macros IDE, find the module name in the Project Explorer and right-click on it.
3. In the popup menu, choose Export (the word Export will be followed by the name of the module).
4. The Export File dialog box will open. Save the module where you can easily find it (it doesn't have to be in a macro directory, but I usually put it in a directory with other macros). Give the exported file a `.vb` filename extension.

The module is now in its own file with a `.vb` extension. You're ready to import it to another project. (But remember, the project will get its own copy of the module, so any changes you make to the module won't automatically be picked up by the project to which you're importing the module.) To import the module:

1. In the Macros IDE, find the project to which you want to import the module. Right-click on the project name and choose `Add` → `Add Existing Item`.
2. The Add Existing Item dialog box will open, and the title bar of the dialog box will also have the name of the project to which you're going to import the module. Make sure this name is the correct project.
3. Find the module you're importing, choose it, and click Open.

The Macros IDE will import the module to your project. Your project will end up with an additional module with the same name as the originally exported module. You're now free to use the module as if it were part of the project—because it *is* part of the project.

Referencing One of Your Own Assemblies

To show you how to make an assembly available to a macro, I'm going to start with a sample macro that I will move over to the main IDE and build an assembly. Then I'll make the assembly available to other macros.



If you already have an assembly and simply want to access it in your macros, all you need to do is copy its DLL to the directory `c:\program files\microsoft visual studio .net\common7\ide\publicassemblies` (substituting the first two directory names in the path if you installed Visual Studio .NET in a directory other than the default). When you open the Add Reference dialog box, you will see the name of your assembly.

As I mentioned in the previous section, if you write some general subroutines or classes that you want to make available to other macros, you can put the subroutines and classes inside an assembly. Here's how you do this. First, suppose the following code is the macro module you wish to make available to other macro projects:

```
Imports EnvDTE
Imports System.Diagnostics

Public Module Output

    Private OutputWin As OutputWindowPane

    Private Function FindOutputPane(ByVal win As OutputWindowPanels, _
        ByVal title As String)
        Dim apane As OutputWindowPane
        For Each apane In win
            If apane.Name = title Then
                Return apane
            End If
        Next
        Return Nothing
    End Function

    Private Sub AddOutputPane(ByVal title As String)
        Dim outwin As Window = DTE.Windows.Item _
            (EnvDTE.Constants.vsWindowKindOutput)
        outwin.Visible = True
        OutputWin = FindOutputPane( _
            outwin.Object.OutputWindowPanels, title)
        If OutputWin Is Nothing Then
            OutputWin = outwin.Object.OutputWindowPanels.Add(title)
        End If
    End Sub

    Public Sub Print(ByVal text As String, _
        Optional ByVal title As String = "Macros")
        AddOutputPane(title)
        OutputWin.OutputString(text & Chr(13))
    End Sub

    Public Sub Clear(Optional ByVal title As String = "Macros")
        AddOutputPane(title)
    End Sub
End Module
```

```
        OutputWin.Clear()  
    End Sub  
  
End Module
```

When I built this module as a macro, I did not add any additional references to the macro project beyond what was already referenced by default. However, as you'll see shortly, I'll nevertheless need to add some references when I transform this into a VB.NET program that will later become an assembly.

Here are the steps to get the preceding module into a VB.NET program, ready for an assembly.

1. In the main IDE, create a new solution by choosing File⇨New⇨Blank Solution; or open an existing solution.
2. Create a new VB.NET class library by choosing File⇨Add Project⇨New Project. In the New Project dialog box, choose Visual Basic Projects in the left tree, then choose Class Library in the Templates list on the right. Type a name and location for your project. Make sure the name is reasonably unique, as this is the name that will appear later on in the Add Reference dialog box when you create a macro to use the assembly you're building.
3. Visual Studio .NET will create a new starter project for you, which will contain a Class1.vb file and an AssemblyInfo.vb file. You can remove the Class1.vb file if you want, as you won't be needing it (at least not for this example).
4. In the Solution Explorer, right-click on the name of the project you just created and choose Add⇨Add Module. In the Add New Item dialog box, make sure Local Project Items in the left tree is highlighted and that Module in the templates list on the right is highlighted. Then type a name for the module. (I recommend using the same name as the module you'll be copying over. For the sample, I used the name Output.)
5. Switch back to the Macros IDE and open the module you'll be using in the assembly. Copy the entire code to the clipboard. (For example, I press Ctrl-A to select all, then Ctrl-C to copy.) Then switch back to the main IDE and replace all the code in the new module with the code from the clipboard. (For example, I press Ctrl-A to select all, then Ctrl-V to paste over the selection.)

At this point, for the most part, your module should be fine as-is, with two important exceptions:

- You will likely need to add some references if you use any of the classes outside of the main `System` namespace.
- The assembly you'll be building doesn't automatically know about the root DTE object, as the macros do, so you'll need a special routine to add it.

Here's how you perform these two actions:

1. For the references, switch back to the Macros IDE; in the Project Explorer, look at the references in the original macro project that contained the code you are

putting in the assembly. Then go back to the main IDE. In the Solution Explorer you'll see there's also a References section to which you can compare the references. If any are missing (which there will be), right-click on the word References and choose Add Reference. Scroll down and click on the missing items. (The names will have a .dll extension, but other than that they will be the same.) One you'll most likely need is the envdte library, which contains the DTE object and class information. When you're finished, click OK.

2. Now you need to make your module recognize the DTE object. To do this, add the following lines inside the module:

```
Private OutputWin As OutputWindowPane
Private TheDTE As DTE
Public Sub Setup(ByVal ADTE As DTE)
    TheDTE = ADTE
End Sub
```

3. Next replace any references to the DTE object with TheDTE. In other words, instead of directly using the DTE object throughout the code, you'll use the TheDTE object, which is a private variable local to the module in the code you just added.

That should finish fixing up the module. Here's the completed module (the changes are shown in bold):

```
Imports EnvDTE
Imports System.Diagnostics

Public Module Output

    Private OutputWin As OutputWindowPane
    Private TheDTE As DTE

    Private Function FindOutputPane(ByVal win As OutputWindowPanels, _
        ByVal title As String)
        Dim apane As OutputWindowPane
        For Each apane In win
            If apane.Name = title Then
                Return apane
            End If
        Next
        Return Nothing
    End Function

    Private Sub AddOutputPane(ByVal title As String)
        Dim outwin As Window = _
            TheDTE.Windows.Item(EnvDTE.Constants.vsWindowKindOutput)
        outwin.Visible = True
        OutputWin = FindOutputPane( _
            outwin.Object.OutputWindowPanels, title)
        If OutputWin Is Nothing Then
            OutputWin = outwin.Object.OutputWindowPanels.Add(title)
        End If
    End Sub
End Module
```

```

        End If
        OutputWin.Clear()
    End Sub

    Public Sub Print(ByVal text As String, _
        Optional ByVal title As String = "Macros")
        AddOutputPane(title)
        OutputWin.OutputString(text & Chr(13))
    End Sub

    Public Sub Clear(Optional ByVal title As String = "Macros")
        AddOutputPane(title)
        OutputWin.Clear()
    End Sub

Public Sub Setup(ByVal ADTE As DTE)
    TheDTE = ADTE
End Sub

End Module

```

Now you're ready to build the assembly. To do this, you need to first set up the assembly information. In the information for this project in the Solution Explorer, notice a file called `AssemblyInfo.vb`. Double-click to open it. You will see several lines that look like this:

```
<Assembly: AssemblyTitle("")>
```

but with various other identifiers in place of `AssemblyTitle`. For the `AssemblyTitle` and `AssemblyDescription`, type a title and description inside the quotes. You can also fill in some of the other information, such as company name and so on. Here's what I filled in and what I left blank:

```

<Assembly: AssemblyTitle("VBMacroUtilities")>
<Assembly: AssemblyDescription("VBMacroUtilities")>
<Assembly: AssemblyCompany("Jeff Cogswell")>
<Assembly: AssemblyProduct("")>
<Assembly: AssemblyCopyright("(c) 2002 Jeffrey M. Cogswell")>
<Assembly: AssemblyTrademark("")>
<Assembly: CLSCompliant(True)>

```

Now here's the really important part: You're going to set up this project so it will create a *strong-named* assembly, which is simply an assembly that includes an encryption key that uniquely identifies the assembly. To add the strong name, first you need to add a line to the `AssemblyInfo.vb` file that you were just modifying. Add the following line to the very end of the `AssemblyInfo.vb` file:

```
<Assembly: AssemblyKeyFile("../..\keyPair.snk")>
```

This is a reference to a file containing a unique key.

Next, to create this file containing a unique key, you need to open up a good old command-prompt window (that is, a DOS window). But don't open the default one that's buried somewhere inside the Start menu of the Windows Desktop. Instead, click on the Start menu, go to the Microsoft Visual Studio .NET group, choose Visual Studio .NET tools, and then click Visual Studio .NET Command Prompt. This is a special version of the standard command prompt that has the path already set up for all the .NET command-line tools.

Once you have the Visual Studio .NET Command Prompt open, change to the directory containing the VB.NET project you created and have been modifying. Make sure you're in the directory that contains the AssemblyInfo.vb file, then type the following command:

```
sn -k keyPair.snk
```

This will generate a file that contains an encryption key pair. You don't need to worry about the contents of the file; the sn program generated a unique pair for you.

Now return to the main IDE. Go ahead and compile the program by right-clicking the project name in the Solution Explorer and choosing Build.

After you're finished building, you need to copy the resulting .DLL file to a special location to enable the macro referencing system to find it. For this you can use either the same Command Prompt window you had opened a moment ago or Windows Explorer, whichever you prefer. Underneath the directory containing your project (the same place you wrote the keyPair.snk file) you will find a bin directory containing the DLL. Here's what I see:

```
VBMacroUtilities.dll
```

Copy this .DLL to the following directory (or modify the first couple of directories in the path if you installed your .NET system elsewhere):

```
c:\program files\microsoft visual studio .net\common7\ide\
publicassemblies
```

Now the .DLL, which is also an assembly, is in place for your macros to use it.

Here's how you can try out the assembly. Switch over to the MacroSIDE, and in one of your macro projects, add a reference to your new assembly. To do so, right-click on the project in the Project Explorer and choose Add Reference. In the Add Reference dialog, find the new one called VBMacroUtilities. Double-click the assembly and click OK to add it to your references.

Now try writing a subroutine that calls a procedure or function in your assembly. If you're using the example I gave you, try this:

```
Sub TryExternalRef()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    VBMacroUtilities.Print("Writing to Output Window!")
End Sub
```

The first line inside the subroutine sets up the module so it knows about the DTE object. The next two lines call subroutines inside the assembly.

Creating Windows and Forms

Since you have access to the .NET framework, you can use the classes in the System.Windows.Forms library. In this and the following section I show you how you can interact with the user through dialog boxes and windows. One particularly exciting topic is how you can create custom windows called forms, complete with the controls you want on them: buttons, listboxes, you name it.

Getting Input from a User

Perhaps the easiest way to get input from the user is with the `InputBox` function. This VB.NET function is built right into the language; it displays a small dialog box containing an Edit control, allowing the IDE user to enter a string. The `InputBox` function returns the user's entry. Here's a sample of the `InputBox` function:

```
Sub SimpleInput()
    Dim name As String
    name = InputBox("What is your name?")
    MsgBox("Your name is " & name)
End Sub
```

The `InputBox` function has four optional parameters after the main string:

- *Title*. The text that appears in the title bar of the message.
- *Default Response*. The text that appears inside the entry Edit box by default when the dialog initially opens.
- *X-Position*. The horizontal position for the dialog box.
- *Y-Position*. The vertical position for the dialog box.

Here's an example using all four of these optional parameters:

```
Sub SimpleInput2()
    Dim name As String
    name = InputBox("What is your name?", "Secret", _
        "Refusal", 100, 100)
    MsgBox(name)
End Sub
```



If the user presses Cancel, `InputBox` will return a string of length 0. This is the same as if the user clears the entry Edit box and then clicks OK.

Creating a Form

Although in the Macros IDE you don't have access to a drag-and-drop designer, as you do in the main IDE when creating a VB.NET or C# program, creating a form is still easy: All you do is create an instance of class `Form`. Here's all you need; first, an `Imports` statement:

```
Imports System.Windows.Forms
```

and then this code:

```
Sub QuickForm()
    Dim f As Form = New Form()
    f.ShowDialog()
End Sub
```

Of course, this code doesn't do much other than create a form. But you can see how it first creates an instance of class `Form`, then it calls `ShowDialog` for the form.

But the form would be better if it had some controls on it. Here's a listing that creates a button, in addition to the form, and adds an event handler to the form. First review the code for an entire module, then I'll explain how it works:

```
Imports EnvDTE
Imports System ' Added this manually!
Imports System.Diagnostics
Imports System.Windows.Forms

Public Module Module1

    Class Handlers
        Sub Button_Clicked(ByVal sender As Object, ByVal e As EventArgs)
            VBMacroUtilities.Print("Click")
        End Sub
    End Class

    Sub ShowForm()
        VBMacroUtilities.Setup(DTE)
        Dim f As Form = New Form()
        f.Width = 400
        f.Height = 300
        Dim b As Button = New Button()
        b.Text = "Click"
        b.Left = 25
        b.Top = 25
        Dim h As Handlers = New Handlers()
        AddHandler b.Click, AddressOf h.Button_Clicked
        f.Controls.Add(b)
        f.ShowDialog()
    End Sub

End Module
```

To use this module, simply call the `ShowForm` subroutine, which exists as a macro. But notice one important aspect to this module: I had to manually add the `Imports System` line, because the Macros IDE didn't add it automatically. The import of `System` makes it easier to directly use the class `EventArgs`, because you don't have to type the fully qualified name, `System.EventArgs`. I also added the `Imports System.Windows.Forms` line, allowing easier access to the various form and control class. And notice in a couple places in the code I'm using the `VbMacrosUtilities` assembly, which I created earlier in this chapter, in the section, "Referencing One of Your Own Assemblies."

Now notice the class called `Handlers` at the top. This class holds a single function called `Button_Clicked`. As you can probably imagine, this function is the code that will handle the click of a button. In the subroutine `ShowForm`, you can see I first set up the output for the `VbMacrosUtilities` assembly, then I create a new form. The next two lines set the width and height of the form.

The next line creates a new button for the form. Note that this line doesn't yet add the button to the form, however; it simply creates the button object. Next I set the text for the button (the button will be labeled "Click"), and then I set the position by specifying the `Left` and `Top` members.

Next I create the instance of the `Handlers` class. Although there are no member variables (the class contains only a single function), I still need to create an instance of the class, because event handlers require an instance, not just a class. Then I *register* the event handler. To register the handler, I call the `AddHandler` function, passing two items:

- *The event I want to catch.* In this case, I want to catch the `Click` event of my button, which is simply called `b`. Thus, the first parameter of `AddHandler` is `b.Click`.
- *The address of the function that will capture the event.* This function must be a function inside an instance, so I don't just pass the function name; I pass the name of the object as well. Thus, I pass `AddressOf h.Button_Clicked`. (The `AddressOf` keyword takes the address of the item that follows, `h.Button_Clicked`.)

As you can see, `AddHandler` doesn't look like a typical function call; its parameters don't have parentheses, an idea that is a throwback to the old BASIC language where built-in functions didn't take parameters (even though `AddHandler` itself did not exist in the old BASIC language).

That's all there is to it: This sample code creates a window with a button, and when you click the button you see the message "Click!" appear in the output window.

Displaying Standard Dialog Boxes

The .NET framework includes classes for displaying standard dialog boxes, such as a font chooser. Here's a sample code that shows the `FontDialog` dialog box, and then sets the fonts accordingly.



To use this listing properly, you need to add the following references: System.Drawing.dll, and VBMacrosUtilities. Also take note of the Imports statements, as you need to add two imports lines as well: the System import and the System.Windows.Forms import.

```
Imports EnvDTE
Imports System ' Added this manually!
Imports System.Diagnostics
Imports System.Windows.Forms

Public Module FontForm

    Class MyFontDialogClass

        Dim f As Form
        Dim b As Button

        Sub Button_Clicked(ByVal sender As Object, ByVal e As
EventArgs)
            Dim fonts As FontDialog = New FontDialog()
            If fonts.ShowDialog() = DialogResult.OK Then
                f.Font = fonts.Font
            End If
        End Sub

        Sub ShowForm()
            VBMacroUtilities.Setup(DTE)
            f = New Form()
            f.Width = 400
            f.Height = 300
            b = New Button()
            b.Text = "Click"
            b.Left = 25
            b.Top = 25
            AddHandler b.Click, AddressOf Me.Button_Clicked
            f.Controls.Add(b)
            f.ShowDialog()
        End Sub
    End Class

    Sub FontDialogTest()
        Dim inst As MyFontDialogClass = New MyFontDialogClass()
        inst.ShowForm()
    End Sub

End Module
```

In this code, I changed the architecture just a bit from the sample in the previous section; I put the ShowForm function inside the main class, and I moved the form and button variables outside of ShowForm and into the class instance itself. That way the Button_Clicked subroutine has easy access to the two variables.

Inside the `Button_Clicked` subroutine you can see how I accessed the font dialog: I first created a new instance of `FontDialog` and then I simply called the `FontDialog` instance's `ShowDialog` function. I then tested the return value of `ShowDialog`, to see if the IDE user clicked OK. If so, I set the new font for the form equal to the `Font` property of the `FontDialog` instance. (Setting the form's font by default sets the font for all the child controls of the form, so the button's font changes as well.)

Here's a list of the standard dialogs available:

- *ColorDialog*. A standard dialog that lets your IDE user choose a color.
- *OpenFileDialog*. A dialog that lets your IDE user choose a file that exists on your computer. Normally, you would use this when you want to prompt the user for the name of a file to open.
- *SaveFileDialog*. A dialog that lets the user specify a new filename anywhere on the computer. Normally, you use this when you want to prompt the user for a name for saving a file.
- *FontDialog*. A dialog that lets the user select a font.
- *PageSetupDialog*. A dialog for setting up the printer.
- *PrintDialog*. A dialog for printing.

Since the focus of this book is macro development, for our purposes here, I'll simply say that these dialog box classes behave the same as in standard .NET development.

Working with System Event Handlers

Often when you write a macro, it's convenient to be able to respond to various activities the IDE user performs. For example, before the IDE user closes a document, you might want to run a macro that records the current date and time to a log file (assuming, of course, that such a macro wouldn't be a violation of privacy).

The Visual Studio .NET Macro system includes a built-in set of events that your macros can respond to; additionally, you can write your own new events. A macro that responds to an event is called an *event handler*. The built-in event handlers have several categories, which I discuss shortly. But first let me explain how to create an event handler.



The event handlers show up only in the macro explorer from within the Macros IDE. They do not show up in the Macro Explorer from within the main Visual Studio .NET IDE. Therefore, to run the event handlers, you must perform the action in the main IDE that triggers the event.

To create an event handler that shows you how to create a handler that takes place whenever the IDE user opens a document, follow these steps:

1. In the Macros IDE, inside either the Class View or Project View, find the project where you wish to create the event handler and expand the project's tree.
2. Double-click the `EnvironmentEvents` module under the project's tree to open it inside the code editor. This module looks just like any other module, except it

has a section of automatically generated code. *Do not modify this code.* You can (and will, if you add an event handler), however, add more code to the module.

3. At the top of the code module are two comboboxes. In the left combobox, find the category for the event you wish to handle. For the example, choose DocumentEvents.
4. After you choose a category in the left combobox, the Macros IDE will fill the right combobox with the events in the chosen category. For the example, choose DocumentOpened.
5. The Macros IDE will automatically insert a subroutine skeleton for the event handler. The subroutine skeleton includes the subheader line, which is important, because the subheader line shows you the object that is passed to the event. For the example, the object passed to the event is the document that was opened (since the event handler is DocumentOpened).

Inside the subroutine, type the code for your handler. For the example, use this code:

```
Public Sub DocumentEvents_DocumentOpened(ByVal Document As _
EnvDTE.Document) Handles DocumentEvents.DocumentOpened
    Dim logfile As System.IO.StreamWriter
    logfile = New StreamWriter("c:\docs.log", True)
    logfile.WriteLine(Document.FullName + " " + Now)
    logfile.Close()
End Sub
```

6. If you need any Imports lines, you can add them now. (Or you can do this step anytime after step 2, where you opened the EnvironmentEvents module.) For the example, add the line `Imports System.IO` after the existing `Imports System.Diagnostics` line.

Now you can try out the event handler macro. For the example, switch back to the main IDE and open any file. (If you have a project open, the easiest way is to click a source code file in the Solution Explorer under the project you have open.) When you do so, you won't see anything happen beyond the file opening; but in the background, the example event handler will run. If you look at the root of the C: drive, you will see a file docs.log. This file will contain a line such as this:

```
C:\temp3.txt 10/2/2002 12:50:56 PM
```

When you open another file, the event handler macro will add another line to the log file. Thus, you will accumulate a log of all files that are opened within the IDE.

Categories of Events

The Visual Studio .NET macro engine has 12 categories of built-in event handlers, which cover most of the common actions that an IDE user can perform. Each category contains several event handlers. Here's the list:

DTE events. These are the events dealing with when the macro engine starts up or shuts down, as well as when the IDE starts up or shuts down. The DTE events are:

ModeChanged. Occurs when the IDE user begins a debug session, and again when the debug session ends. (Note that the Visual Studio .NET online help claims this event also occurs when a build session begins or ends, and when the program being built runs. That is not the case: The ModeChanged event occurs only when beginning and ending a debugging session.)

OnBeginShutdown. Occurs when the IDE user shuts down the Visual Studio .NET IDE. Specifically, the event occurs when the shutdown process begins.

OnMacrosRuntimeReset. Occurs when the macro engine resets. The main use for this is when you have custom event handlers (as opposed to handlers for the built-in events): inside this event handler you would reattach the event handlers.

OnStartupComplete. Occurs after the IDE user starts up the Visual Studio .NET IDE and after the startup process is complete. Remember, when you open either the Macro Explorer or the Macros IDE, you can choose which of your macro projects are loaded or unloaded. Those that are loaded will reload the next time you start up the main IDE. Thus, if you have an OnStartupComplete handler, it will run the next time you start up the IDE.

Document events. These are the events dealing with documents, such as opening and closing. The Document events are:

DocumentClosing. Occurs just before the IDE user closes a document.

DocumentOpening. Occurs when the IDE user opens a document. Specifically, this event occurs before the IDE opens the user's chosen document. Note that the first parameter to the DocumentOpening event handler is a String type called DocumentPath. This parameter is actually the full path and filename of the document, not just the path.

DocumentOpened. Occurs when the IDE user opens a document. Specifically, this event occurs after the IDE opens the user's chosen document.

DocumentSaved. Occurs *after* the IDE user saves a document.

Window events: These events take place as the IDE user manipulates the document windows in the main IDE. These events do not occur in response to the IDE user manipulating the tool windows. The Window events are:

WindowActivated. Occurs when any window in the IDE receives the focus.

WindowClosing. Occurs when the IDE user closes a document window. Note that this event occurs *before* the DocumentClosing event occurs.

WindowCreated. Occurs when the IDE user opens a document window. This event occurs *after* the DocumentOpening event, but *before* the DocumentOpened event. Thus, the order is (1) DocumentOpening, (2) WindowCreated, and (3) DocumentOpened.

WindowMoved Event. Occurs when the IDE user moves or resizes a document window. This event really only matters when the IDE user has chosen the MDI environment from the Environment:General section of the Options dialog box. If the IDE user instead has chosen Tabbed documents, he or she cannot move the windows, thus the WindowMoved event has no effect.

Task List events. The task list is a window with checked items that you can use for keeping track of a to-do list relating to your project (although, really, you can add any item, not just project-related items). The IDE also adds items to this list when the IDE user builds a project and there are errors. The IDE adds a line for each error. The Task List events are:

TaskAdded. This event occurs after a new item is added to the task list, when the IDE user builds a project and there are errors; the errors appear in the task list. Or, this event can happen when the IDE user manually adds an item to the task list. The event handler receives an instance of the `TaskItem` class, which contains information about the task that was added. The following is a sample TaskAdded event handler:

```
Public Sub TaskListEvents_TaskAdded(ByVal TaskItem As _
    EnvDTE.TaskItem) Handles TaskListEvents.TaskAdded
    MsgBox("TaskAdded: " + TaskItem.Description)
End Sub
```

TaskModified. Occurs only after the IDE user modifies an item in the task, not when he or she deletes or adds a task. The event handler receives an instance of the `TaskItem` class, which contains information about the task *after* it was modified.

TaskNavigated. Occurs when the IDE user double-clicks on an item in the task list that refers to compiler errors or warnings or other to-do information in the task list pertaining to the source code. When the user double-clicks such an item, the TaskNavigated event will occur, *then* the IDE will switch focus to the source code, highlighting the line to which the task item refers. For example, if the IDE user builds a project and there is an error in the code, the IDE will add an item to the task list describing the error. When the user double-clicks the error message in the task list, the TaskNavigated event will occur and then the IDE user will see the line with the error appear in the source code. The line will be highlighted.

TaskRemoved. Occurs when an item is removed from the task list. The event occurs just before the IDE removes the task. However, when the IDE user builds a project, and there are existing error and warning messages in the task list, the IDE removes the existing error and warning messages, but the TaskRemoved event does not occur. The event occurs only when the IDE user manually removes an item from the task list.

Find events. The IDE contains a Find in Files dialog box, which the IDE user can use to find search strings through multiple files. The Find category of events actually contains only one event, FindDone:

FindDone. Occurs after the IDE user performs a Find in Files operation. The event occurs after the search is finished. The event handler receives a `vsFind-Result` enumeration, which simply tells the status of the completed Find in Files operation—whether the search item was or wasn't found, a replace operation found the searched item, a replace failed because no search items were found, the search did not complete, or the search resulted in an error.

Output window events. The output window in the IDE is a multipurpose window that displays the output of various activities, such as the output from a build or from a debugging session. At the top of the output window is a drop-down listbox that contains a list of panes. When you choose an item in the list, the output window switches to output for the item you chose. The output category consists of the following events:

PaneAdded. This event occurs when a new output window is added, but only in response to macros and add-ins that add an output pane. This event does not occur when the IDE adds an output pane (for instance, when the IDE user begins a debugging session, the IDE creates a Debug pane).

PaneClearing. This event occurs when an output pane clears, for example, when the IDE user right-clicks inside the output window and chooses Clear All.

PaneUpdated. This event occurs when the output in a pane changes, whether text is added or the text is cleared. This includes when the IDE user builds a project. Note that this event occurs every time a line of text is added to the output pane, which can cause numerous `PaneUpdated` events to occur.

Selection events. This event category contains only one event, which deals with the item is currently selected within the IDE. Think of the selection as the combination of the window that currently has the focus (whether it's a document window or a tool window) and, within that window, the item that is currently active.

OnChange. When the user clicks either on a different item within the window that currently has the focus or on a different window altogether, the `OnChange` event occurs. Note, however, that this event handler does not have any parameters, thus the handler does not receive any information on the selection. To find out what is currently selected, inspect the `DTE.SelectedItems` object. The following sample code demonstrates this.

```
Public Sub SelectionEvents_OnChange() Handles SelectionEvents.OnChange
    Dim item As SelectedItem
    Dim mystr As String = ""
    If DTE.SelectedItems.MultiSelect = True Then
        For Each item In DTE.SelectedItems
            mystr = mystr + item.Name + Chr(13)
        Next
        MsgBox(mystr)
    Else
        MsgBox(DTE.SelectedItems.Item(1).Name)
    End If
End Sub
```

Build events. The Build events occur when the IDE user builds a project or solution. Even if a project is up-to-date, these events still occur. They also occur when the IDE issues a clean command. The Build events are:

OnBuildBegin. Occurs when the IDE user builds a project. If the user builds an entire solution containing multiple projects, only one OnBuildBegin event will occur. To find out which project begins, see the OnBuildProjConfigBegin event in this list.

OnBuildDone. Occurs when a build process is complete. Again, if the IDE user builds an entire solution, this event will occur only once even though the solution contains multiple projects.

OnBuildProjConfigBegin. Whenever a build begins, this event occurs. This event contains the name of the project, the name of the configuration (such as Debug or Release) and the name of the solution configuration. If the user chooses to build an entire solution, the OnBuildProjConfigBegin event occurs prior to the build of each project in the solution.

OnBuildProjConfigDone. Occurs after each project build. For multiple projects within a solution, this event occurs after each project in the solution builds. An event handler for this event receives the same information as the handler for the OnBuildProjConfigBegin event.

Solution events. The Solution events occur in response to various activities involving the solutions. They are:

AfterClosing. Occurs after a solution closes.

BeforeClosing. Occurs before a solution closes.

Opened. Occurs after a solution opens.

ProjectAdded. Occurs after the IDE user adds a project to the solution.

ProjectRemoved. Occurs after the IDE user removes a project from the solution.

ProjectRenamed. Occurs after the IDE user renames a project in the solution.

QueryCloseSolution. This event is interesting because the handler receives a Boolean variable that the handler can change: If the handler determines the solution should not close after all, the handler can return `True` to cancel the closing of the solution.

Renamed. Occurs after the IDE user renames the solution.

Debugger events. These events deal with interaction with the debugger. The Debugger events are:

OnContextChanged. Occurs when the user (or a macro) changes the current process, program, thread, or stack.

OnEnterBreakMode. Occurs when the debugger encounters a breakpoint.

OnEnterDesignMode. Occurs when the IDE returns from debug mode.

OnEnterRunMode. Occurs when the process being debugged begins to run, either initially or after a breakpoint.

MULTIPLE HANDLERS FOR A SINGLE EVENT

What if you have multiple macro projects open and you write an event handler in each project for the same event? The IDE will simply call each of the event handlers, one after another. However, there is no guarantee as to the order in which they will be called. This is because the events use Visual Studio .NET *delegates*, which handle the business of calling event handlers. A delegate calls the handlers in the order in which they are added. But in the macros, you do not have control over this order. Therefore, if you are handling the same event in multiple macro projects, do not tie them together, with one relying on the other happening first.

OnExceptionNotHandled. Occurs when the program being debugged encounters an exception that was not handled by the program.

OnExceptionThrown. Occurs when an exception is thrown. It will take place before an `OnEnterBreakMode` event.

Moving Forward

In this chapter I discussed the different ways that you can write a macro that interacts with the user. This includes creating forms and writing to the main IDE's output window. Additionally, I took you through the steps of building an assembly called `VBMacroUtilities` that writes to the output window. Be sure to follow the steps I outlined, as you will need this assembly for many of the macros in the remainder of the book.

Initially, the next chapter diverts focus from macros to address some .NET architecture. Halfway through the chapter, I return to the macros to show you how the projects and solutions coexist, and how you can manipulate them with your macros.

TEAMFLY

Just Enough .NET Architecture

By virtue of the fact that you're reading a book on developing macros in Visual Studio .NET, it's probably safe to assume that you know at least some .NET programming. Nevertheless, for those of you not that familiar with .NET, in this chapter I talk about the architecture of .NET, to help get you up to speed. For those of you proficient in .NET programming, this chapter will serve to show you how .NET fits into the macro development world.

Getting to Know Microsoft .NET

Since .NET was introduced, around 1999, many people have tried to define exactly what it is. Some think it's an online subscriber-based service, much like MSN, where users can log in and check their email. (In fact, there's good reason for this: When you visit the MSN page, Microsoft now includes a big welcome to .NET.) But to most software engineers, .NET is the latest and greatest layer that sits atop the Windows operating system. In the past, there were OLE and COM (which are still very much a part of Windows) and something called Windows DNA (which many programmers ignored; and to this day most still don't have a clue what Microsoft was attempting with it).

In fact, .NET is something new altogether. It's an entire framework that provides:

- Managed applications, which means the .NET framework takes care of object management, including the deletion of objects that are no longer used. In other words, C++ programmers don't need to worry about deleting their objects,

provided the objects are *managed objects*. This also means the operating system watches over the security of systems, ensuring safe execution of code that might not be trusted. Finally, managed applications also run in Microsoft Intermediate Language (MSIL), which is a highly optimized language.

- A rich class library.
- Easy ways to create Web services.
- A solution to the problem of different programs expecting different versions of the same DLL.
- A software development kit to aid in .NET programming.
- A development tool for creating .NET programs, called Visual Studio .NET.
- Flexibility in deploying applications that target multiple platforms. Included in this is just-in-time compilation, which means the runtime will compile the program from MSIL to native code either at installation time or on the fly at runtime.
- Support for multiple language development. Since code is compiled to an intermediate language, developers can use any language for which there exists an MSIL compiler. Such languages include C++, C#, and Visual Basic .NET.

In order to provide for all these features, Microsoft has supplied two layers:

- Common Language Runtime (CLR)
- .NET Class Library

The CLR is the heart of the .NET system. It is the layer that sits on top of the operating system and provides the .NET features for managed applications. When you execute a .NET program, the CLR handles the just-in-time compilation and the execution of the program, along with memory management, security, and thread management. When a program is a .NET application, it is, in actuality, an application that targets the CLR.



If you're familiar with Java, the CLR is very much like the Java Virtual Machine (JVM) in concept: When you run a Java program, it runs on top of the JVM, while the JVM handles memory management and garbage collection, along with other features such as thread management. Also, when you write Java programs, they are compiled on the fly using a just-in-time compilation process, just like .NET programs.

The .NET class library is composed of a rich set of classes that provide features such as string handling, console output, file handling, and window and form management.

In the sections that follow I describe the CLR and the .NET Class Library.

Common Language Runtime

Common language runtime is the foundation of the .NET Framework. It provides the fundamental services that you would normally expect to find in a runtime system, such as memory allocation and thread management. But unlike other runtime systems, the common language runtime also includes a number of basic data types. This is an

important feature that helps simplify cross-language development. Prior to .NET, if you created a library of classes and functions, the language in which you developed the library made a difference to how the library was used by other languages.

For example, if you wrote a set of classes and standalone functions in C++ and saved them to a library, but wanted to access the classes and functions in either Pascal (using, for example, Borland Delphi) or Visual Basic, you were in for quite a job. For starters, strings are stored differently in different languages; and other issues might arise as well, such as whether the different languages use the same standard for floating-point numbers, and even the same byte order for integers. If there were differences, you had to twist your data before calling the functions, and then do another twist once you had the results of the function. To top it all off, the order in which variables are passed to the functions also differs. (That's why, if you did Windows programming in C or C++ in earlier days, you may have noticed that frequently you had to throw the keyword "pascal" in front of some function calls, whereas Pascal and C/C++ pass their function parameters in the opposite order.)

But beyond the basic data types, classes are a nightmare. Previous versions of Visual Basic, Delphi, and Microsoft Visual C++ all store classes differently, with different virtual table mechanisms. (Incidentally, at the time of this writing, Borland has announced a version of Delphi that is fully compatible with .NET.) Older versions of VB didn't even support inheritance. I could go on and on about the differences.

To resolve these issues, Microsoft gave us the CLR. Unlike the Java Virtual Machine, which provides cross-platform development but requires that programs all be written in a single language, Java, the CLR lets you choose your language. For example, you can choose C++, the new C#, or the newest Visual Basic, and, soon, Borland Delphi's Object Pascal.

Presently, you're pretty much stuck targeting Windows for a platform; but that might soon change as well, as developers come to recognize the importance of working along with Microsoft Windows (whether they like Windows and Bill Gates or not) and begin to work on .NET-compatible libraries on Unix and Linux. (For more information on .NET support under Linux, go to www.dotnet.za.net/. At the top is a link for .NET on Linux.)

.NET Class Library

The .NET class library is a full-featured library that contains many different namespaces, each serving a separate purpose. A reference to the entire class library would easily fill an entire volume; therefore, here I list only some of the more commonly used groups of classes. You can find the entire reference in the online help if you open the contents and drill down to Visual Studio .NET→.NET Framework→Reference→Class Library.

Here, then, are some of the namespaces you'll find, along with some of the more useful classes:

System namespace. This contains basic types, including a root `Object` class and class versions of the fundamental MSIL types. Basic type classes include `Int32` (which is a C++ class; in VB.NET, you use `Integer`) and `String`. This namespace also has the `Console` class, which allows you to write to the console. (Macros, however, don't have a console window, so the `Console` class isn't very useful in the macro world.)

System.Data namespace. This namespace contains classes for accessing databases.

System.Drawing namespace. This namespace includes types relating to graphics. Classes include `Bitmap`, `Brush`, `Color`, `Font`, `Icon`, `Image`, `Point` (a structure that holds simply X and Y values), `Rectangle` (a structure that holds X, Y, Width, and Height values), and `Size` (a structure that holds Width and Height values). Because the `Point`, `Rectangle`, and `Size` structures are quite useful, classes in other namespaces occasionally use these structures in their member functions. If you call any of these member functions, you need to include a reference to the `System.Drawing` namespace.

System.IO namespace. This namespace handles file input and output. It includes many useful classes for various types of file I/O, such as a general-purpose `File` class, as well as an extremely handy `Directory` class for manipulating directories (something ANSI C++ is lacking), and a `Path` class for manipulating path names stored as strings (the `Path` class only manipulates strings; it doesn't actually perform any file input or output). Other useful classes include `MemoryStream`, `StreamWriter` (for writing text to a file), and `StreamReader` (for reading text from a file). Note that if you work with the classes in the `System.IO` namespace, you'll probably notice two classes called `TextWriter` and `TextReader`. These two classes are actually abstract classes that serve as a base for `StreamWriter` and `StreamReader`, respectively, along with several other classes. If you open the online help for either `TextWriter` or `TextReader`, you can see a list of the derived classes. Also, if you want to open a `StreamReader` or `StreamWriter` instance, use the `File` class's `OpenText` method or `CreateText` method, respectively.

System.Net namespace. It's hard to imagine a macro that would need access to the Internet, but this namespace is here for that purpose, filled with several Net-related classes. Of course, the namespace is useful in .NET programs other than macros that need access to the Internet. The `System.Net` namespace also includes another namespace, `System.Net.Socket`, which has several classes for dealing with low-level sockets.

System.Web namespace. This namespace includes classes that operate at a much higher level than the classes in the `System.Net` namespace, providing access to Web-based information. This namespace is part of the greater world of ASP.NET.

System.Xml namespace. This namespace provides the tools for parsing XML files. If you're an XML guru, you'll be pleased to know the namespace is complete, with regard to the various XML standards.

Packaging Your Software: Assemblies and Manifests

One of the fundamental features of .NET is that it uses *assemblies*, which are really just DLLs or executable files, but packaged with additional .NET information to what's normally in a DLL or executable. Further, in the case of DLLs, assemblies are stored in a way that makes versioning more manageable.

An assembly is the fundamental unit of execution in .NET; it contains the Microsoft intermediate language code, along with the startup code that invokes the CLR, passing the MSIL code to the CLR for compilation and execution.

In addition to the MSIL code, an assembly contains a *manifest*, the metadata inside an assembly that describes the assembly. This metadata includes:

- *General information.* The manifest can contain general information such as the name of the assembly and the version number.
- *Security information.* The manifest can contain permission requests, such as the permission to write to the hard disk, but the computer on which the assembly runs may have a permission setup that forbids the application to do so. The assembly will then not be able to run.
- *The files that make up the assembly.* Assemblies can be split up into multiple modules, with a single file containing the manifest, and the code existing in separate files called modules. To learn more about this feature, open the .NET online help and drill down to Visual Studio .NET→.NET Framework→Programming with the .NET Framework→Programming with Application Domains and Assemblies→Building a Multifile Assembly. (And while drilling down, you'll probably find some interesting topics along the way you might want to take the time to read.)
- *A list of other assemblies this assembly depends on.* As with any library or executable, an assembly can depend on other assemblies.

Looking at an Assembly

If you want to look at the information contained in an assembly, you can use the *ildasm* program (which stands for Intermediate Language Disassembler). This program provides you with a graphical user interface (GUI) that displays information about an assembly, including the metadata in the assembly's manifest and the code in the assembly. By default, the *ildasm* program is installed in the C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin directory.

When you run the *ildasm* program, you first see a window from which you can choose File→Open. In the Open dialog box, you then choose an assembly, normally either an .EXE or .DLL file. When *ildasm* opens the assembly, you will see a treeview, with the root node showing the full path to the assembly. Under the root is the word MANIFEST. If you double-click on MANIFEST, a new window will open showing the metadata for the assembly.

Under MANIFEST are the namespaces available in the assembly, along with any top-level items that are not in a namespace. Under each namespace are the items contained in the namespace, including classes and functions. Under classes you will see the members of the class. To access code items such as functions and member functions, double-click an item and see its MSIL code.

Organizing Multiple Versions of an Assembly

If you've ever installed a software package that's a year or two old, only to have the software's installation program overwrite a DLL in your Windows or Windows\System32 directory, replacing a newer DLL with an older DLL, then you're aware that, currently, DLL versioning is a nightmare in Windows. Once the newer DLL is gone and an older one is in its place, very possibly newer software that was already on your computer will no longer function—definitely not a fun situation. And if you reinstall the newer software to restore the new DLL, it's also possible the older software will no longer function!

Microsoft has ended this DLL nightmare by incorporating a clever versioning technique, whereby different versions of a DLL will be installed in separate directories on your system. Here's how it works: In the Windows directory is a directory called assembly, and under it is one called Global Assembly Cache (GAC). Inside the GAC directory is a separate directory for each DLL. And inside each of these DLLs is a separate directory for each *version* of the DLL. For example, if you have an assembly called MainUtilities, and over the past year you have created two different versions of MainUtilities, version 1.0.0.0, and version 1.2.0.0, you can install both DLLs into the GAC. The GAC directory would then have a directory called MainUtilities, and under that would be two directories, one for version 1.0.0.0 and one for version 1.2.0.0, each containing the respective DLL.



If you try to traverse the GAC directory using the Windows Explorer, you will quickly find that you can't. The reason is that, technically, you're not supposed to know (or, at least, not supposed to care) how the directory structure that I just described is organized. Instead, when you install .NET, you get a Windows Explorer add-in (called a *shell extension*) that displays the assemblies, not in a hierarchy of directories, but rather as a list, including the name and the version, anytime you try to look at any directory in Windows\assembly. So if you want to see the actual directory structure, you need to use the DOS command prompt.

So that you don't have to be concerned about the directory structure, the .NET framework includes a utility that installs an assembly into the GAC for you, while maintaining the integrity of the directory structure. This utility is called gacutil.exe, and by default it's installed in the C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin directory.



If you're interested in using the gacutil program to install your own assembly into the global assembly cache (or remove your assembly from the cache), make sure you create the assembly with *strong naming*. If you're curious how to do this, refer to "Referencing One of Your Own Assemblies" in Chapter 4, "Macros That Interact with the User."

The gacutil program has a lot of command-line options, but I find for most work I only need a couple of the options. Here's a sample line to install an assembly into the GAC:

```
gacutil /i MainUtilities.dll
```

This command will uninstall a particular version of an assembly:

```
gacutil /u MainUtilities,version=1.2.0.0
```

This command will uninstall *all* versions of an assembly (so be careful):

```
gacutil /u MainUtilities
```



If you used the DOS command prompt to dig down into the assembly directory, and your prompt is sitting inside the directory of an assembly that you're trying to remove, the gacutil program will not be able to remove the assembly: You will see the error message, "The process cannot access the file because it is being used by another process." To prevent this error, change to some directory other than that of the assembly you're trying to remove.

.NET and OLE/COM

In the past, much of Windows revolved around Component Object Model (COM) components. Originally, Microsoft created Object Linking and Embedding (OLE), and from there developed OLE 2. But the developers decided they didn't want versions (I'm not making this up) so they dropped the 2 and just called OLE 2, OLE. They then broke it up into architectural layers and called the bottommost layer COM. Today, OLE typically refers to compound documents and the sharing of data between applications. Other examples of COM include ActiveX controls (which used to be called OCX controls).

The COM topic is enormous, warranting entire books written about it. But at the heart of COM are *interfaces* and *objects*. When you work with COM, you write code that conforms to a particular interface to be used in a particular situation. For example, you might provide a class that is to be used as a GUI control in a pre-.NET Visual Basic program. In order for the older Visual Basic to access your class, your class must provide a certain set of member functions. These member functions comprise the *interface*. In the COM and OLE world, people like to say that this interface is a *contract* between an object and its users (the users are usually called *clients*).

The interface concept is somewhat complex in that you can create a class that supports multiple interfaces, allowing your class to be used in multiple situations. To use your class, a client requests a pointer to the interface it needs, then the client uses this interface to call the member functions in your class.

The COM concept dates back to the early 1990s when Microsoft decided it was the way of the future. Programmers everywhere began building COM components and sending them out with their applications. One look at the `oleview` program (by default installed in `C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools`) will show you just how many COM components are installed on your computer.

Every COM component includes a type library, which is data that describes the types inside the COM component. Typically, COM components live as a DLL (although they can live as EXEs as well) and this type library is embedded right inside the .DLL or .EXE file as a resource. (If you see other extensions such as .OCX, it's really a DLL with a different filename extension.) As an alternative to putting the type information inside the file as a resource, you can put the type information in a separate file with a .TLB extension (which stands for type library). Normally, you would put this .TLB file in the same directory as the .DLL file.

Globally Unique Identifiers

Then, to make life a bit more complicated, Microsoft implemented an entire structure for the COM system that makes use of *globally unique identifiers*, or GUIDs for short. (Most people pronounce GUID either “goo-id” and “goo-eed.”) A GUID is a 32-digit hexadecimal number; Microsoft's idea was to create an algorithm that could generate a GUID guaranteed to be unique. That is, if you and I simultaneously run a program containing the algorithm, Microsoft assures us that you and I will *not* get the same number. And supposedly you and I can run the program over and over and over and never see the same number. (In part, it works like this: The algorithm takes the current time, along with some numbers that seem to uniquely represent the computer on which the algorithm runs, which pretty much guarantees uniqueness.)

Each COM component, then, gets a GUID, as does each interface. So if somebody defines an interface (such as one to be implemented by COM components that are to serve as controls on a pre-.NET Visual Basic window), then that person will assign a GUID to the interface. Then if you and I each build a COM component that implements the interface (that is, if we both create a window control), we will each assign a GUID to our program and we will state that our program implements the interface given by the interface GUID.

As you can imagine, all this has resulted in piles upon piles of GUIDs (most of which come out of Microsoft's development group). If you open up your Registry using the `regedit` program and drill down to `HKEY_CLASSES_ROOT\CLSID`, you will see tons of COM classes that are installed on your computer, all listed by GUID. If you then drill down to `HKEY_CLASSES_ROOT\Interface`, you'll find all the COM interfaces your computer knows about, also arranged by GUID. And if that's not enough, if you look at `HKEY_CLASSES_ROOT\TypeLib`, you'll find all the type library information your computer knows about, arranged (you guessed it) by GUID.

Building an Assembly from a COM Component

So now that all these GUIDs are on your computer, Microsoft has changed its mind. OLE and COM are not the way of the future. Instead, today, it's .NET. So what to do about all those COM components floating around? Microsoft decided to make .NET

compatible with COM. To accomplish this, it built a utility that lets you create an assembly based on the type information in the COM component's .DLL, .EXE, or .TLB file. Ultimately, the assembly will simply call into the COM component. The end result is that you, the .NET programmer, can use the COM component as if it's just another class in an assembly. The actual COM aspects are then hidden from you.

If you want to create an assembly based on a COM component, you can use the `tlbimp` program (which stands for Type Library Import). This program is by default installed in `C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin.` It takes as input a file containing type library information for a COM component. Thus, the program is either a .DLL or an .EXE representing a COM component that contains its own type library information as a resource, or it's a .TLB file. As output, the program generates a .DLL that is a valid .NET assembly.

Here's an example of how to use the `tlbimp` tool. According to the `oleview` program, my computer has an ActiveX COM control called `ioRdyCtl.ocx`, which is apparently related to the Iomega tools that I installed for working with my zip drive. This COM control is installed in `C:\Program Files\Iomega\Common`. If I want to create an assembly that can access this control, here's a command that I can run from the DOS prompt:

```
tlbimp "C:\Program Files\Iomega\Common\ioRdyCtl.ocx" /namespace:Iomega
```

The first parameter is the name and location of the COM file. The second parameter is optional; it's a namespace under which I would like all types in the assembly to be. When I ran this line, I ended up with a file called `IORDYCTLLib.dll`, which is a valid assembly.

You can also supply a version number for the assembly if you want to, using another command-line option:

```
tlbimp "C:\Program Files\Iomega\Common\ioRdyCtl.ocx" /namespace:Iomega  
/asmversion:1.0.1.0
```

After running this line, you can look at the resulting assembly using the `ildasm` utility. When I tried it on the Iomega assembly, I saw a namespace called `Iomega` (since that's what I requested in the `tlbimp` command-line options). Under the `Iomega` namespace, I saw all the classes and types available there. These were originally classes and types in the COM control; now they're available as .NET classes.

To test out the assembly, you can create a new Visual Basic or C# program in the main IDE and add a reference to the assembly (by right-clicking `References` under the new project in the Solution Explorer, clicking `Browse`, and selecting the assembly file). Then create a new subroutine, type the namespace name, then a period, and you'll see a popup list of all the classes and types in the namespace. If you see this list, you know everything worked.

Macro and Add-in Models

One of the major challenges in understanding the models behind the Visual Studio .NET macro and add-in development is simply sorting out all the classes and namespaces. If you have wandered through the online help, you've probably seen many

classes and namespaces that may seem somehow interrelated, yet somehow separate. In this and the following sections, I try to clear up the general organization of all the namespaces and classes, as well as explain to you how they all fit together and how to use them.

Visual Studio Packages

Visual Studio .NET is built on several libraries that Microsoft calls *packages*. These packages form the heart of Visual Studio .NET and exist as COM objects. You can see the list of packages if you open up the regedit program (from the Start Menu, choose Run and type regedit). Expand the HKEY_LOCAL_MACHINE, then SOFTWARE, then Microsoft, then VisualStudio, then 7.0. Under the key Packages you'll see a whole list of COM objects. Each one serves a separate purpose.

Some of these packages are useful to macro and add-in developers, and include features such as the control and manipulation of solutions in Visual Studio .NET, or control and manipulation of the various tool windows and menus. Microsoft has made these more useful features available for macro and add-in development. Other packages are less useful to macro and add-in developers.

To make the more useful packages available for macro and add-in programming, Microsoft has implemented assembly versions of the packages. The packages themselves are simply COM objects; therefore, the folks at Microsoft ran the `tlbimp` utility to create the assembly versions (at least we can assume that's how they were created). The end result is that you can use these assemblies to control the Visual Studio .NET IDE from your macros and add-ins.

Visual Studio Project Types

Depending on which products you have installed (such as Visual C++, C#, and Visual Basic .NET), your copy of Visual Studio .NET will recognize various project types. If you want to dig through the Registry further, you can find out exactly what types of projects your installation supports. Under the HKEY_LOCAL_MACHINE, then SOFTWARE entries, then Microsoft, and finally inside VisualStudio, then 7.0, you'll find a key called Projects. This key lists the project types available under your installation. Each project type has a GUID associated with it; this GUID is used when you access the `Project` objects in the macros and add-ins. Here's a list of some of the common items you're likely to see, along with their GUIDs:

- Project Converter {059D6162-CD51-11d0-AE1F-00A0C90FFFC3}
- Visual Studio Deployment CAB Project {0b7288ca-6892-4441-925d-34d99f5c97bd}
- Visual Studio Deployment Project {54435603-dbb4-11d2-8724-00a0c9a8b90c}
- Visual Studio Deployment Setup Project {5443560c-dbb4-11d2-8724-00a0c9a8b90c}
- Visual Studio Deployment Tier Project {5443560d-dbb4-11d2-8724-00a0c9a8b90c}
- Visual Studio Deployment Merge Module Project {5443560e-dbb4-11d2-8724-00a0c9a8b90c}

- Visual C++ Project {8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}
- Visual Basic Project {F184B08F-C81C-45f6-A57F-5ABD9991F28F}
- C# Project {FAE04EC0-301F-11d3-BF4B-00C04F79EFBC}

When you use a macro to obtain a `Project` object, the object's `Kind` property will contain a GUID for the type of project. Thus, if you see the GUID {8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}, from the preceding list you can see that you have a Visual C++ Project. Here's a macro that prints out the name of each project in the current solution and the GUID the project's type:

```
Sub ListProjects()
    ' Use the VBMacroUtilities assembly from Chapter 2.
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim proj As Project
    For Each proj In DTE.Solution.Projects
        VBMacroUtilities.Print(proj.Name & " " & proj.Kind)
    Next
End Sub
```

After you know the type of project, you can obtain the `Project` object's `Object` property; this property contains an instance of a COM object that's specific to the type of project. For example, a Visual C++ `Project` object will have a COM object that holds information specific to Visual C++ Projects. In contrast, a Visual Basic `Project` object will have a COM object that holds information specific to Visual Basic projects. (In the case of the preceding macro, the COM object would be `proj.Object` inside the `For` loop.)



One way you can find out the project GUID for a particular project is to simply open the project file (the file of any extension ending in `proj`, such as `.csproj`) in a text editor. Once you open the file, you'll see an XML file with a line starting `ProjectGUID`. The Project GUID will follow, surrounded by double quotes.

When you access the projects using the `Project` object, but don't go further into the `Project` object's `Object` property, Microsoft regards your work as taking place in the *General Extensibility Model*. Under this model, you can access general information about a project, but no information that's specific to the language. (For example, a C++ project might be written to access the Microsoft Foundation Classes (MFC). Information specific to only C++ projects, then, would mention whether the project uses MFC, whereas information specific to VB.NET projects would have no need for information about MFC.) When you start accessing the information stored in the `Object` property of the project, Microsoft says you are working in the *language-specific model*.

When you're looking at the `Kind` property to determine the type of object, you probably don't want to memorize all these GUIDs or have to type them in (can you say "typos?"). Fortunately, the .NET framework has defined some constants for you that you can use in place of the GUIDs—well, at least, it has three constants defined for you:

- `PrjKind.prjKindVBProject`. The GUID for a Visual Basic project.
- `PrjKind.prjKindCSharpProject`. The GUID for a C# project.
- `PrjKind.prjKindVBAPProject`. The GUID for a macro project running in the Macros IDE. I talk about this in the next section, “Macros IDE Automation Model.”

These are the only three constants defined by default. The reason is that these are defined in a library called `VSLangProj` that’s specifically for these three types of projects. As for constants for the other GUIDs, Microsoft wasn’t so kind as to supply them. However, you can easily create them yourself. But, really, the only one you’re likely to need is the Visual C++ Project kind. The reason is that, at present, the only kinds of projects Microsoft has made accessible from a language-specific perspective are the Visual Basic, C#, Macro/VBA, and C++. For the other types of projects, you get back the GUIDs in the list; that said, the object you get back isn’t immediately accessible—unless you use a small trick, which I discuss next.

Accessing Project Types Not Supported by the Macros IDE

I’m only going to give you basic guidance for using the trick to access the other project objects; and from there, Microsoft certainly doesn’t support this, but I’m providing you with this trick if you want to use it at your own risk. Here’s what you need to do:

1. Find the GUID of the type of project you want to access. (You can start with the list of GUIDs I gave you earlier; the rest are in the Registry as I described.)
2. Once you have the GUID, locate the GUID in the `Projects` key in the main `VisualStudio/7.0` key in the Registry.
3. Under the key you will find another key called `AddItemTemplates`; under that is another key called `TemplateDirs`. Under that key you’ll find another GUID. This is a GUID for a package.
4. Go up to the `Packages` key in the same `VisualStudio/7.0` section and locate the GUID for the package. Under that key you’ll find a key called `SatelliteDll`. This key contains both a `Path` and a `DllName` that together contain the name of a COM server.
5. Run the `tlbimp` program to import the COM object as an assembly. This assembly will then contain a class for the object.

So far so good, but there’s one piece missing: You need to know the class name of the object stored in the `Project` object’s `Object` property.

One easy way to determine the class name is to call the `TypeName` function, which is a function built into VB.NET. Here’s an example:

```
MsgBox (TypeName (proj .Object))
```

When you run this line, a message box will open showing you the name. When I ran this for the `Setup` project, I saw the name `IVsdDeployable`, thus I knew the object class in this case was `IVsdDeployable`, which corresponds to a `Setup` project.

Accessing the Project Object

In this and the next two sections, I describe the architecture behind the projects. To illustrate this discussion, I show you some macros that retrieve information from the projects. (For a full discussion of projects—including how to modify them using your macros—refer to Chapter 9, “Manipulating Solutions and Projects.”)

To gain access to the Project object, you start with the DTE object’s Solution property to get the Solution object. The Solution object contains information about the currently loaded solution; it also contains a Projects property that is a collection of all the projects. Each item in this collection is an object of type Project. The Project object contains only information that is generic, not specific, to the language; thus, Microsoft considers it as part of the general extensibility model.

The Project object includes a ProjectItems property that contains the items the project holds, including source files and the folders that contain the source files. Here’s a macro that lists all the items:

```
Sub ListProjectItems()
    ' Use the VBMacroUtilities assembly from Chapter 2.
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim proj As Project
    Dim obj As Object
    For Each proj In DTE.Solution.Projects
        VBMacroUtilities.Print(proj.Name & " " & proj.Kind)
        Dim pItem As ProjectItem
        If Not proj.ProjectItems Is Nothing Then
            For Each pItem In proj.ProjectItems
                VBMacroUtilities.Print(" " & pItem.Name)
            Next
        End If
    Next
End Sub
```

Here’s a sample output for a project called MyProgram. This is a C++ project, and therefore the output shows .cpp and .h files along with the folder names:

```
MyProgram {8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}
  MyProgram.cpp
  AssemblyInfo.cpp
  stdafx.cpp
  stdafx.h
  ReadMe.txt
  Source Files
  Header Files
  Resource Files
```

In the two sections that follow, I show you how to access the project types that are supported by the IDE, Visual Basic, C#, and C++.

Accessing Visual Basic and C# Projects

To access language-specific information about Visual Basic and C# projects, you need to add to your project a reference to the `VSLangProj` assembly, if it's not already referenced by your Macro project. This library is an assembly version of the `vslangproj.tlb` type library, which gets installed by default in the directory `C:\Program Files\Common Files\Microsoft Shared\MSEnv`.

When you have a `Project` object, and you take its `Object` property, you will get an instance of `VSPProject`. `VSPProject` is defined in the `VSLangProj` assembly. This object contains information specific to Visual Basic and C# projects. To determine if the `Project` object refers to a Visual Basic or C# project, check its `Kind` property, as in the following Macro code:

```
If (proj.Kind = VSLangProj.PrjKind.prjKindVBProject) _
Or (proj.Kind = VSLangProj.PrjKind.prjKindCSharpProject) Then
    Dim vsproj As VSLangProj.VSPProject
    vsproj = proj.Object

End If
```

Inside this `If`-block, after you get the `proj.Object` object, you can use it to access information about the project. Here's an extended version of this same macro code that lists a great deal of information about a project:

```
Sub ShowProjectInfo(ByVal proj As Project)
    ' Use the VBMacroUtilities assembly from Chapter 2.
    If (proj.Kind = VSLangProj.PrjKind.prjKindVBProject) _
Or (proj.Kind = VSLangProj.PrjKind.prjKindCSharpProject) Then
        Dim vsproj As VSLangProj.VSPProject
        vsproj = proj.Object
        VBMacroUtilities.Print("Project: " & proj.Name)
        VBMacroUtilities.Print("References:")
        Dim ref As VSLangProj.Reference
        For Each ref In vsproj.References
            VBMacroUtilities.Print(ref.Path)
        Next
        If (proj.Kind = VSLangProj.PrjKind.prjKindVBProject) Then
            VBMacroUtilities.Print("Imports:")
            Dim i As Integer
            For i = 1 To vsproj.Imports.Count
                VBMacroUtilities.Print(vsproj.Imports.Item(i))
            Next
            VBMacroUtilities.Print("")
        End If
    End If
End Sub
```

And here's an example of a macro that calls the `ShowProjectInfo` function:

```
Sub PrintVBandCSProj()
    ' Use the VBMacroUtilities assembly from Chapter 2.
```

```

VBMacroUtilities.Setup(DTE)
VBMacroUtilities.Clear()
Dim proj As Project
For Each proj In DTE.Solution.Projects
    ShowProjectInfo(proj)
Next
End Sub

```

When I ran the `PrintVBandCSProj` macro on a solution that contained several VB.NET and C# projects, I saw this output for one of the projects (the `VBMacroUtilities` project that I use throughout this book).

```

Project: VBMacroUtilities
References:
C:\WINNT\Microsoft.NET\Framework\v1.0.3705\System.dll
C:\WINNT\Microsoft.NET\Framework\v1.0.3705\System.Data.dll
C:\WINNT\Microsoft.NET\Framework\v1.0.3705\System.Xml.dll
C:\WINNT\Microsoft.NET\Framework\v1.0.3705\envdte.dll
C:\WINNT\Microsoft.NET\Framework\v1.0.3705\Microsoft.Vsa.dll
C:\WINNT\Microsoft.NET\Framework\v1.0.3705\office.dll
C:\WINNT\Microsoft.NET\Framework\v1.0.3705\System.Windows.Forms.dll
Imports:
Microsoft.VisualBasic
System
System.Collections
System.Data
System.Diagnostics

```

You can see all the assemblies that the project references, as well as all of the imports. Since references and imports are language-specific items (C++ doesn't use either, for example), the reference and import information is not present in the main `Project` object. Instead, they're both in the `VSPProject` item, which is language-specific for VB.NET and C#.



In the `ShowProjectInfo` macro, I fully qualified all the type names. For instance, I specified `VSLangProj.VSPProject`. If you prefer to simply specify `VSPProject` rather than the fully qualified name, you can add an `imports` statement at the top of the macro's module, as shown here:

```
Imports VSLangProj
```

Then you can declare variables such as this:

```
Dim vsproj As VSLangProj.VSPProject
```

Accessing C++ Projects

To access information specific to C++ projects, you need to reference the C++ project library. There are three assemblies pertaining to C++ projects, but in this section I only

describe one of them, the `VCProjectEngine` library. I discuss the other two in Chapter 9, “Manipulating Solutions and Projects.”

To get to the C++ project information, start with a `Project` object, check if the `Kind` property specifies a C++ project, then take the `Project` object’s `Object` property, and cast it to a `VCProject` class. To check the `Kind` property, you probably want to create a constant variable containing the GUID for a C++ project, like so:

```
Const prjKindVCProject = "{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}"
```



The class you’ll be using in this section is called `VCProject`. One of the C++ project assemblies is also called `VCProject`. However, the class `VCProject` is actually a part of the `VCProjectEngine` assembly, not the `VCProject` assembly.

Next is an extended version of the `ShowProjectInfo` subroutine from the preceding section, modified to support C++ projects. Notice I included the new constant `prjKindVCProject` as well. To use this subroutine this time, instead of fully qualifying the names, I included an import statement at the top of the module:

```
Imports Microsoft.VisualStudio.VCProjectEngine
```

Now here’s the subroutine:

```
Const prjKindVCProject = "{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}"

Sub ShowProjectInfo2(ByVal proj As Project)
    ' Use the VBMacroUtilities assembly from Chapter 2.
    If (proj.Kind = VSLangProj.PrjKind.prjKindVBProject) _
    Or (proj.Kind = VSLangProj.PrjKind.prjKindCSharpProject) Then
        Dim vsproj As VSLangProj.VSProject
        vsproj = proj.Object
        VBMacroUtilities.Print("Project: " & proj.Name)
        VBMacroUtilities.Print("References:")
        Dim ref As VSLangProj.Reference
        For Each ref In vsproj.References
            VBMacroUtilities.Print(ref.Path)
        Next
        If (proj.Kind = VSLangProj.PrjKind.prjKindVBProject) Then
            VBMacroUtilities.Print("Imports:")
            Dim i As Integer
            For i = 1 To vsproj.Imports.Count
                VBMacroUtilities.Print(vsproj.Imports.Item(i))
            Next
            VBMacroUtilities.Print("")
        End If
    ElseIf proj.Kind = prjKindVCProject Then
        Dim vcproj As VCProject
        vcproj = proj.Object
    End Sub
```

```

VbMacroUtilities.Print("Project: " & proj.Name)
VbMacroUtilities.Print("Configurations:")
Dim cfg As VcConfiguration
For Each cfg In vcproj.Configurations
    VbMacroUtilities.Print(cfg.Name)
    VbMacroUtilities.Print("  BrowseInfo:" & _
        cfg.BuildBrowserInformation)
    VbMacroUtilities.Print("  Uses MFC:" & _
        cfg.useOfMfc)
    VbMacroUtilities.Print("  Is .NET app:" & _
        cfg.ManagedExtensions)
Next
End If
End Sub

```

This subroutine accesses the configuration info that's specific to C++ projects. For each configuration, I print out the name of the configuration, whether the Build Browse Information selection is turned on, whether the configuration uses MFC, and whether the configuration uses Managed Extensions (and is, therefore, a .NET application).

Here's an updated macro subroutine that uses the `ShowProjectInfo2` subroutine:

```

Sub PrintAllProjInfo()
    ' Use the VbMacroUtilities assembly from Chapter 2.
    VbMacroUtilities.Setup(DTE)
    VbMacroUtilities.Clear()
    Dim proj As Project
    For Each proj In DTE.Solution.Projects
        ShowProjectInfo2(proj)
    Next
End Sub

```

When I ran the `PrintAllProjInfo` subroutine on my own projects, here's the information I saw for one of them:

```

Project: MyProgram
Configurations:
Debug|Win32
  BrowseInfo:False
  Uses MFC:0
  Is .NET app:True
Release|Win32
  BrowseInfo:False
  Uses MFC:0
  Is .NET app:True

```

Notice that the `Uses MFC` item printed out a number, 0, rather than a Boolean value of `True` or `False`. The reason is that the `useOfMFC` property holds one of three numbers: 0 means the project doesn't use MFC; 1 means the project uses MFC and links to the MFC static library; and 2 means the project uses MFC but links to the MFC dynamic library.

Macro IDE Automation Model

The Macros IDE has its own automation engine, which at first seems a little strange to people. As you've certainly noticed, the Macros IDE looks very similar to the main IDE. That's because, although it's a separate program, it uses many of the packages used underneath the main IDE.

The Macros IDE is actually the program called `vsaenv.exe`. Under the default installation, you can find it in `c:\Program Files\Common Files\Microsoft Shared\VSA\7.0\VsaEnv\vsaenv.exe`. Since this program is essentially a knock-off of the main IDE, but with certain features stripped away and other features added, the program also has an entire section in the Registry that lists its packages and projects, among other things. The Registry entry, a sibling to the main IDE's Registry entry, is called VSA. (The letters VSA in the Registry entry and in the executable filename `vsaenv.exe` stand for Visual Studio for Applications.)

As to the project types, the Macros IDE lets you create only one kind, a Visual Studio for Applications VB.NET Project, that is, a macro project. This project has a GUID of {13B7A3EE-4614-11D3-9BC7-00C04F79DE25}, but since you don't want to have to remember that (or type it in), you can use `PrjKind.prjKindVBAProject`.

Just as in working with the main IDE, when you acquire a `Project` object, you can check if its `Kind` property equals `PrjKind.prjKindVBAProject`. If so, then you have a macro project. But the way you acquire a `Project` object for the Macros IDE is slightly different from how you get one for the main IDE: Instead of using the root `DTE` object, you use the `DTE` object that belongs to the Macros IDE. You can get this using the object `DTE.MacrosIDE`.



If you look at the properties for the main `DTE` object, you'll see a property called `Macros`, which in turn has a `DTE` object—that is, there's an object `DTE.Macros.DTE`. This object is not, however, the `DTE` for the Macros IDE. It's actually just the original main `DTE`. The `Macros` property represents the macro recorder, not the Macros IDE. And the `Macros` property's `DTE` object is just its parent object, which is the main `DTE` object.

Here's a sample macro that obtains the projects in the Macros IDE and writes out their names and their `Kind` value:

```
Sub ListMacroProjects()
    ' Use the VBMacroUtilities assembly from Chapter 2.
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim proj As Project
    For Each proj In DTE.MacrosIDE.Solution.Projects
        VBMacroUtilities.Print(proj.Kind & " " & proj.Name)
    Next
End Sub
```

Here's a sample line of output from this macro:

```
{13B7A3EE-4614-11D3-9BC7-00C04F79DE25} Samples
```

You can see the GUID is the same as the one mentioned earlier. Following the GUID is the name of the project.

Like the Visual Basic projects in the main IDE, when you have a macro project in the Macros IDE, you can cast the `Project` object's `Object` property to a `VSPProject` object to obtain more information about the project from a language perspective.

Moving Forward

In this chapter I introduced the .NET architecture, then applied it from the perspective of writing macros and add-ins. Regardless of the type of product you are developing, probably the three most important aspects to understand about the .NET architecture are:

- What the CLR is and how it fits into the picture
- What an assembly is
- Which classes are available in the .NET framework

In the next chapter, which begins Part II, I introduce the high-level architecture of add-ins, then explain what they are and how they fit into the system. In Chapter 7, I take you through the steps of building an add-in.

TEAMFLY

PART

Two

Enhancing Visual Studio

Introducing Add-ins

An add-in is a powerful way to enhance the Visual Studio .NET IDE. The idea behind an add-in is to enable developers to create custom enhancements to the IDE. When you create an add-in, it becomes fully integrated with the IDE, giving the IDE user the illusion that the add-in is not a separate component from the IDE itself. In this chapter I provide you with much of the background information you will need to build add-ins, illustrated with a couple of add-ins to get you going; then, in Chapter 7, I show you sample code for many of the concepts I lay out in this chapter.



If you're like most programmers, you don't read this type of book cover to cover, instead preferring to skip around. However, I do recommend that you read this chapter *before* any of the remaining chapters in Part II.

About Add-ins

An add-in is a DLL that you create and that Visual Studio .NET loads into its own process. Your add-in can communicate with the Visual Studio .NET IDE, to access the different parts of the IDE, such as the solutions and projects, the code editor, and the tool windows. Moreover, the add-in can respond to events that take place in the IDE.

If this is starting to sound similar to a macro, that's because they are similar. An add-in has access to all the automation features and objects that a macro does; this means that pretty much any task you perform in a macro you can also do in an add-in. In spite

of all these commonalities, an add-in has one major distinction from a macro: An add-in is compiled, then the main IDE loads it as a DLL; a macro is not compiled, and Visual Studio .NET's macro engine interprets the macro on the fly. Furthermore, because an add-in is compiled into a DLL, you can distribute it without having to distribute the source code. That's not the case with macros; when you distribute a macro you are distributing the source code. For this reason, if you're worried about your intellectual property, then certainly you want to build an add-in instead of a macro.

Another advantage of add-ins over macros is that you have your choice of language in which to develop them, whereas macros are limited to Visual Basic; add-ins can be in any .NET language (that is, any language that can be compiled to Microsoft Intermediate Language). And, if you're brave, you can create an add-in that's not a .NET assembly, because the add-ins are COM controls at heart. However, I don't recommend that, so I don't show you how to do this in this book.

As I just said, when you create an add-in, you are given the choice of language to use. Depending on the languages you have installed, your choices will be: C++, C#, and Visual Basic .NET. There aren't any real advantages or disadvantages to any of these three languages, since all three are compiled to the same intermediate language, MSIL; therefore, pick the language based on what you prefer and which one you know best.



All the .NET languages are available for building add-ins, so I give examples in all three: C++, C#, and VB.NET.

When you create an add-in for Visual Studio .NET, one of your goals should be to make the add-in seem like a natural part of Visual Studio .NET. Here are some ways you can accomplish this goal:

- Stick to a common look and feel. If you have a dialog box, don't make custom buttons and graphics that look nothing like the buttons and graphics in the Visual Studio .NET IDE.
- Make use of the IDE's GUI elements, including:
 - Tool windows
 - Menus
 - Toolbar buttons
- Use the Options dialog box to allow for customization of your add-in.
- Integrate it directly to the IDE. By this I mean that if, for instance, your add-in makes modifications to the IDE user's source code, make the modifications right in the document window, not just to the file itself. And, certainly, don't launch an external code editor.

Here are some other goals you might strive toward:

- Make your add-in flexible. Let the IDE user customize it.
- Make your add-in easy to use. This is Windows, after all: don't require the user to memorize complex keystrokes and commands.
- Give the user flexibility for installing and uninstalling the add-in.



If you've poked around in the .NET Framework SDK, you may have come across a rather interesting program called DbgCLR, which by default gets installed in `C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\GuiDebug`. This is simply a standalone version of the debugger that's integrated to the main Visual Studio .NET IDE. Since this program is built on many of the standard Visual Studio .NET packages, you might expect that you could create add-ins and macros for this tool. Unfortunately, that's not the case. You can't create add-ins or macros for the DbgCLR tool.

Startup and Shutdown

When you create and install an add-in, you can specify whether you want the add-in to be present the next time you start up the main Visual Studio .NET IDE. Your add-in has the capability to respond to events, and some of these events deal with the system starting up and shutting down. Therefore, if you want your add-in to do any specific work when the IDE starts up (such as opening some files and initializing some objects) you can respond to the event for the system startup. Similarly, if you want to do some cleaning up before the IDE shuts down, you can respond to the event for the system shutdown.



The event you will receive for the system startup happens after the system is finished starting up—that is, after the IDE has finished loading its various packages and tool windows. You will see, for example, the Start page appear in the Web browser just as you receive the system startup event. As for the shutdown process, your add-in will receive a shutdown event just as the IDE begins its main shutdown procedure. However, this one is a little touchy: If you watch closely, you will see some shutdown activities take place before your add-in receives its shutdown event. For example, I saw the browser window inside the IDE close before I received my event.

Alternatively, you can configure an add-in so that it does not start up automatically. In this case, the IDE user must manually start up or shut down the add-in. And just as with the startup and shutdown of the IDE, your add-in can also respond to the startup and shutdown events for the add-in itself.



As you would expect, if your add-in is registered to start up when the IDE starts up, your add-in will receive one event for the startup of the IDE and another event for the startup of the add-in itself. Specifically, your add-in will first receive the event for when the add-in starts up, followed by the event for the IDE starting up. When shutting down, a similar process takes place, but in the reverse order: First you will receive an event that the system is beginning to shutdown; then you will receive an event that your add-in is shutting down.

The Visual Studio .NET documentation calls the startup of your add-in the *connection* and the shutdown of your add-in the *disconnection*. There's a good reason for this: First, Visual Studio .NET uses the COM system to load the DLL containing your add-in. Then Visual Studio .NET sends a connection event to your add-in. Therefore, there's a moment during which your add-in is actually loaded but you have not yet received the event from Visual Studio .NET. Nevertheless, when your add-in receives the connection event, technically it's already loaded.

When you write your add-in, you will see that the names of the events I just described are as follows:

OnConnection. Occurs when the add-in starts up.

OnDisconnection. Occurs when the add-in finishes.

OnStartupComplete. Occurs when the IDE finishes its own startup process.

OnBeginShutdown. Occurs when the IDE is about to shut down.

Invoking Your Add-in

Visual Studio .NET provides you with different ways to invoke your add-in. First, when you create the add-in using the Add-in wizard, you can specify that you want your add-in to start up automatically when the main IDE starts. Or, you can specify that you don't want the add-in to start up automatically, in which case the IDE user can start the add-in manually using the Add-in Manager. (The Add-in Manager is a dialog box accessed through Tools⇨Add-in Manager that lets the IDE user load and unload add-ins.)

Once your add-in is loaded, it can receive various commands from IDE actions, either through IDE events or when the IDE user issues a command to your add-in. To issue a command, the IDE user has three choices:

- Click a button on a toolbar that runs a command inside your add-in.
- Choose a menu item that runs a command inside your add-in.
- Issue a command directly through the command tool window.

But this raises a question: If you allow the IDE user to access your add-in through a button on a toolbar or a menu item, where does the button or menu item come from? The answer depends on which type of button or menu item. There are two kinds:

- A button or menu item that invokes a command inside your add-in.
- A button or menu item that loads your add-in.

In the case of the first, the add-in itself can install the button or menu. But what about the second one? How can the add-in have a button or menu that starts up the add-in? The programmers invoke a trick to make this happen: The add-in automatically loads itself the first time; thereafter, the IDE user can load the add-in by clicking a button or menu item. Here's how it works:

1. When you write your add-in, you create it so that, when it is installed, it starts up automatically.

2. You code your `OnConnection` event handler to install a button or menu item that loads the add-in. Further, you code the add-in so it removes the automatic startup feature.
3. Optionally, you can include code that checks whether the button or menu is showing, and if so, disables the button or menu item. Then you include code in the `OnDisconnect` event that reenables the button or menu item.

At first, this process might seem a bit counterproductive, but consider this scenario for which the three steps would work:

1. The IDE user installs your add-in, which is registered to start when the IDE starts. The user then starts the IDE.
2. The add-in loads at the IDE startup and checks for a button or menu item. There isn't one, since this is the first time the add-in has run, so the add-in adds the button or menu item. The add-in now registers itself to *not* start up automatically. And, if you took the optional step 3 in the process, you would now disable the menu item or button you just added, since the add-in is presently running and there's no reason (yet) for a menu item or button that starts the add-in.
3. The user does some work and then shuts down the IDE.
4. The user later restarts the IDE. Now, though the add-in hasn't started yet, the button or menu item is present that will start it. The user clicks the button or chooses the menu item and the add-in loads. And, if you took optional step 3, the add-in sees that the button or menu is already there and disables it so the user cannot click it again.

From the IDE user's perspective, this is all simple: The first time he or she installed the add-in, it ran automatically. After that, there was a button or menu item to start the add-in. Later, when you create an add-in, you will see that the Add-in wizard gives you the opportunity to install a menu item on the Tools menu bar. This menu item performs precisely the steps I just outlined.

Interacting with the IDE

When you create an add-in, you have access to the root `DTE` object of the IDE. This means you can interact completely with the IDE, whether you're dealing with the main IDE or the Macros IDE. Through this object, you can obtain other objects, such as the `Solution` object, through which you can modify solutions and projects. Or you can interact with the Output window, in which you can print useful messages to the user.

Unlike with macros, however, you do not simply access the `DTE` object directly via the name `DTE`. Instead, you receive a reference to the `DTE` object in your `OnConnection` event, and you save the reference into a variable (usually using a class member variable). Then, throughout your code, instead of accessing `DTE` directly, you use the class member variable.



Fortunately, when you use the wizard to create an add-in, the wizard automatically inserts code to your class for the `OnConnection` event handler. This code saves the reference to the `DTE` object into a member variable. Therefore, if you use the wizard, you don't need to write the code yourself to save the `DTE` object.

Through the root `DTE` object, you have full access to the IDE, including the toolbars, all the windows (both document windows and tool windows), the current solution, and the commands. Here's the complete list of item categories your add-in can interact with:

- Solutions and projects, which represent the entire solution and its projects.
- The build objects, which are used to automate compiling and building of projects and solutions.
- Code editor objects, which allow you to select code, modify it, and perform other editing tasks automatically.
- Code definitions, which gives you have access to the actual code engines (this topic is discussed in Chapter 9, "Manipulating Solutions and Projects").
- Tool and document windows, including specific tool windows such as the output window and the Solution Explorer.
- Commands, which are those the IDE can run. (More on this later in this chapter, in the section "Add-ins and the Command System.")
- Debugger objects, which automate the use of the debugger.
- Events from the IDE other than the startup and shutdown events.

Because the add-in has access to the root `DTE` object, you can create add-ins that do the same work as a macro. Say, for example, you have a macro that works well, but for which you don't want to distribute the source code: In this case, you can reimplement the source code as an add-in. And because you have flexibility in the language you use, you can use `VB.NET` just as you do in the macros, meaning you'll have to make very few changes to your code. (But, remember, in your macro you probably accessed the root `DTE` object directly, so you will have to modify your code to use the reference to the `DTE` object passed in to your `OnConnection` event handler.)

Recall from Part I, which focused on macros, that I took one of my more useful macros and made an assembly out of it. I then put the assembly in the `publicassemblies` directory, which made the assembly accessible to any macro. In doing so, I had to take the source code from my macro and move it to the main IDE, into its own `VB.NET` project. This means that, when you develop your code to interact with the IDE, you have the following choices:

- You can write macro code.
- You can write an assembly that the macro and add-ins can access.
- You can write an add-in.

How you implement your code is up to you, and the choice you make will depend on how you want it to work with other macros and add-ins.

When dealing with the IDE, be aware of one important distinction between macros and add-ins: When you run a macro, the IDE pauses and waits for the macro to finish. This is not the case with an add-in. The IDE loads an add-in and then periodically sends events to the add-in as the IDE user interacts with the IDE (which might include interacting with GUI elements created by the add-in, such as toolbar buttons, menu items, and tool windows). No multithreading is taking place: If your add-in goes into a long loop, the IDE will freeze up until your loop finishes, just as with macros. However, the point is, your add-in, once loaded, remains present, and can receive multiple events and commands.

Creating Custom Options Pages

One handy feature of add-ins is the capability to create custom pages for the Options dialog box (accessed by the IDE user through Tools⇨Options).

There are two similar reasons you might want to add a page to the Options dialog box: First, you might want to give the user a way to configure your add-in itself; second, you might want to give the user a way to configure the data your add-in is controlling or managing. (Or you might even want to do both.)



You might be tempted to use your own approach to allowing your users to configure your add-in. It might, for example, seem nice to have a menu item that opens up a special dialog box for configuring your add-in. In general, however, this is a bad idea. Better to give the IDE users the feeling that your add-in is an integrated part of the IDE. Think about it: Many of your users probably will be programmers themselves, as well as power users, hence not so easily fooled (or impressed, for that matter); and probably they'll be sticklers for a common look and feel. Therefore, your best bet is to use the Options dialog box.

As a consequence of being able to write add-ins that work in the main IDE or the Macros IDE or both, you can also set up pages for the Options dialog box in the Macros IDE, as well as the main IDE. But before you design a page for the Options dialog box, make sure you spend a little time becoming familiar with the general look of the Options dialog box. Note, for example, that the left side of the Options dialog box lists categories of options, arranged hierarchically. You will want to create your options pages to match this structure.

When you create an options page, you create and register an ActiveX control that the Options dialog loads. This ActiveX control contains the form that the Options dialog displays. You can put controls on the form and write code that interacts with the form, just as you would for any dialog box. Through this code you can save the options the IDE user chooses through your Options page.

Next you add an entry to the Registry under your add-in detailing the hierarchy for the options categories. (In Chapter 17, "Supercharging Visual Studio .NET," when I show you how to create an options page, I also show you how to register the ActiveX control.)

Creating Tool Windows

If you want your users to be able to interact with your add-in using more than buttons and the Options dialog box, you can create tool windows with your add-in that are fully integrated to the IDE. Besides the practical uses, tool windows also give your add-in a more integrated feel, as if it's an integral part of the IDE.

When you create a tool window, the IDE user will be able to manipulate it the same way as any other tool window, by moving it around, docking it, and making it float. Furthermore, your tool window will automatically take on a tab when the IDE user drags it onto another window. All this happens automatically; you don't have to do any programming, because the features are built right into the Visual Studio .NET IDE. The end result is that your add-in appears to be an integral part of the IDE.

When you work with tool windows, you can also allow users to open up a property window, which you can fill with various properties pertinent to your add-in. For example, if your add-in is a list of files arranged in a treeview, you can allow the IDE user to click on one of the files in the list, after which the property window will fill with information about the particular file. This is just like the standard use of the property window inside the main IDE: The IDE user can right-click on many different items inside the IDE and in the popup menu choose Properties, which opens the Properties window.

Add-ins and the Command System

One of the fundamental parts of the main IDE is the command system. When you choose View⇨Other Windows⇨Command Window, a tool window opens that contains a prompt in the form of a greater-than symbol (>). In this window, you can type commands to interact with the IDE. The commands are all categorized according to this format: name.command or name.commandgroup.command. Recall that in Chapter 1, "All about Macros and Add-ins," I gave you a macro called GetAllCommands that opens a window and displays all the commands; here I give you a different version of the macro: it writes the commands to the output window. To use this, you need to use the VBMacroUtilities assembly that I described in Chapter 3. Here's the code for this simple macro (remember, this is a macro, not an add-in, although you could use the code in an add-in):

```
Sub ShowCommands ()
    Dim cmd As Command
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    For Each cmd In DTE.Commands
        If Not cmd.Name Is Nothing Then
            VBMacroUtilities.Print(cmd.Name)
        End If
    Next
End Sub
```

This macro will list all the commands. One command, for example, is `File.Close`, which, as its name implies, closes a document. If you open the command window and type the following:

```
File.Close
```

whichever file is active in the document window will then close.

Through this command window the IDE user can also interact with your add-in. Your add-in can register new commands, which the IDE user can type in. When the IDE user types a command for your add-in, your add-in receives an `Exec` event, along with the name of the command the user typed. Through your `Exec` event handler, you can respond to various commands.



When you create an add-in, the `Exec` event handler is created automatically for you by the Add-in wizard. You can then simply add to the existing code for this event handler to respond to various commands.

In addition to interacting directly through the command window, your add-in can set up a toolbar button and attach a particular command to the button; or you can do the same with a menu item. Then when the IDE user clicks the toolbar button or chooses the menu item, your `Exec` handler will receive the name of the command.



When your add-in starts up, you can register the commands you plan to process. Once you do so, the commands you register will become part of the command list. Your command will then be part of the `DTE.Commands` list in the macros, as well as the command list in the Configuration dialog box under the Commands tab, in the Add-ins category.

Wizards

Another feature you can implement with your add-ins is a dialog box with a series of step-by-step pages to guide the IDE user through some process, such as configuring your add-in the first time the add-in runs. Such a dialog box is called a *wizard*, and it behaves just like the wizards you use when you create an add-in or other application. You can launch the wizard from within your add-in (such as in response to a command) or you can set up the IDE so the wizard appears in the New Project or New File dialog boxes. You have full control over what the wizard looks like; and like the rest of an add-in, you can use C++, VB.NET, or C# to create the wizard.

But when you create a wizard, before it can appear in the New Project or New File dialog boxes, you have to create a special file with a `.vsz` filename extension. You then store this file with other `.vsz` files. (An alternative is to create a different type of file that instead has a `.vsdir` extension. This includes additional information beyond what you put inside a `.vsz` file, including information about an icon you would like to appear inside the New Project or New File dialog box.)

In Chapter 12, “Creating Project Wizards,” when I show you how to create a wizard, I also show you how to create the `.vsz` or `.vsdir` file and where to put the file.

Add-ins Are COM Components

As you are writing an add-in in one of the .NET languages, and ultimately building an assembly, be aware that, actually, you are building a COM component with which the main IDE and Macros IDE can interact.

Remember that a COM component implements an interface, which simply means that the COM component has several available functions that a caller expects to see. (In the case of an add-in, the caller is the IDE.) By convention, COM interface names always start with an “I,” for interface. And when you create a COM component (including an add-in), you register it in the Registry using various GUIDs (which I explained in Chapter 5, “Just Enough .NET Architecture.”)

When you create an add-in, you must implement an interface called `IDTExtensibility2`. (A lot of interfaces provided by Microsoft end with a 2, to indicate they’re newer versions of an interface Microsoft designed some time ago for earlier products.) By implementing the `IDTExtensibility2` interface, you create a class in your add-in that has the member functions defined in `IDTExtensibility2`. The IDE will then call your member functions at certain times (in other words, these functions are event handlers). Some of the member functions I define here you’ve seen earlier in this chapter:



At this juncture, I’m giving you an overview of all these member functions and the interfaces I describe later in this section. I show you how to use them in detail in Chapter 7.

OnAddInsUpdate. When an add-in is either added or removed, the IDE will call the `OnAddInsUpdate` function for all add-ins currently loaded.

OnBeginShutdown. When the IDE begins to shut down, it will call this function for all add-ins that are loaded.

OnConnection. When an add-in is loaded, the IDE will call this function only for the add-in that was loaded. (Add-ins do not receive this event for other add-ins.)

OnDisconnection. When an add-in is being unloaded, the IDE will call this function only for the add-in being unloaded.

OnStartupComplete. When the IDE is finished starting up, it will call this function for all add-ins that are loaded. Naturally, this will affect only the add-ins that are registered to start up when the IDE starts.

In addition to the `IDTExtensibility2` interface, an add-in usually implements a second interface called `IDTCommandTarget`. This interface allows the add-in to respond to commands. (When you create an add-in using the Add-in wizard, your code will automatically be set up for both the `IDTExtensibility2` and `IDTCommandTarget` interfaces.)



In case you’re curious, and though the online help doesn’t explicitly state this, my best guess is that the letters “DT” in the interface names stand for Development Tools, the same as in DTE (Development Tools Extensibility). And the “I,” of course, stands for Interface.

The `IDTCommandTarget` interface has two functions; therefore, for your add-in to implement this interface, it must supply these two functions, which the IDE will call. They are:

- Exec.** The IDE calls this function when it wants to issue a command that your add-in is to execute.
- QueryStatus.** The IDE calls this function to ask your add-in if a command is available. Your `QueryStatus` function will return a combination of enumerations specifying whether the command is `Unsupported`, `Supported`, `Enabled`, or `Invisible`.



With the `QueryStatus` function, there are two other possible enumerations your function can return. These are the oddly named `Latched` and `Ninched`. After searching far and wide on the Internet (including Microsoft's MSDN site), I finally found out what these two items mean (and which few people seem to use): `latched` indicates the command is an on/off toggle and that it's currently in the on state; `ninched`, which seems to have little application to commands, indicates the command is really a set of commands covering multiple commands, and that the different commands are in different states. (Microsoft gave this analogy on its Web site to explain this: Consider some text selected in a word processor document, some of which is bold and some of which is not bold. The state of the text as a whole being bold is "ninched.") Good thing English is a growing, evolving language; I was unable to find "ninched" in the dictionary!

With add-ins, interfaces find their way into two other areas: wizards and pages in the Options dialog box. Wizards implement the `IDTWizard` interface. This interface only has a single function called `Execute`. In the code for your wizard, inside the `Execute` function, you write the code that displays the wizard.

As for Option dialog box pages, the interface in question is called `IDTToolsOptionsPage`. This interface has five functions:

- GetProperties.** This function deals with returning a set of properties.
- OnAfterCreated.** The IDE calls this function when a custom Tools option page gets created.
- OnCancel.** The IDE calls this if the IDE user clicks the Cancel button in the Options dialog.
- OnHelp.** The IDE calls this if the IDE user clicks the Help button in the Options dialog.
- OnOK.** The IDE calls this if the IDE user clicks the OK button in the Options dialog.

Although I just covered four interfaces, typically you'll have three different COM objects. The first is the actual add-in object, which implements the `IDTExtensibility2` and `IDTCommandTarget` interface. The second is the wizard, which implements the

IDTWizard interface. And the third is the Options dialog box page, which implements the IDTToolsOptionsPage. Of course, if you're building an add-in, you don't have to include a wizard or an Options dialog box page; it's your choice as to which features you want to include.

For the IDE to be able to use the three COM components I just mentioned, each component must be registered. When you use the Add-in wizard, the resulting project takes care of registering the add-in COM component. (In Chapters 12 and 17, when I show you how to build a wizard and Options page, respectively, I also show you how to register them.) With add-ins, however, there are actually two different stages of registration: registering the add-in so that Visual Studio .NET knows about it and registering the COM component so COM system knows about it.

It is at this point that things can get a little touchy. When you run the Add-in wizard, the wizard itself registers your add-in so that the Visual Studio .NET IDE knows about it. (And, optionally, the wizard will register the add-in with the Macros IDE if you choose to make the add-in available to the Macros IDE.) The registration of the COM component itself, however, takes place each time you rebuild the add-in. Moreover, when you use the wizard to create an add-in, you also end up with an add-in Setup project that you can build and then use for installing the add-in. That's handy if you want to ship your add-in to other users. But here's the problem: What if you download the source code for an add-in and you get the add-in project but not the Setup project? I bring this up because that's exactly what happens if you download the sample add-in projects from the Microsoft site, at <http://msdn.microsoft.com/vstudio/downloads/samples/automation.asp>. In that case, you need to do the registration yourself, manually. (The online help makes a brief mention of the process, but suggests it's just nice to know. The truth is, it's *important* to know; furthermore, the online help gives only half the instructions).

In the next section, "Creating an Add-in," you'll learn to use the wizard to create an add-in. The wizard creates some starter code for you, which includes a GUID for the add-in. In the event you don't have a Setup project and weren't fortunate enough to run the wizard, if you have the GUID and the name of your add-in, and you come up with a description, you can use the following macro to register your add-in with Visual Studio:

```
Sub RegisterForIDE()
    Dim guid As String
    Dim name As String
    Dim friendlyname As String
    Dim description As String
    guid = InputBox("Enter the GUID, including the braces { }")
    If guid = "" Then
        MsgBox("No changes made to registry.")
        Exit Sub
    End If
    name = InputBox("Enter the formal name")
    If name = "" Then
        MsgBox("No changes made to registry.")
        Exit Sub
    End If
    friendlyname = InputBox("Enter the friendly name")
    If friendlyname = "" Then
```

```

        MsgBox("No changes made to registry.")
        Exit Sub
    End If
    description = InputBox("Enter a description")
    If description = "" Then
        MsgBox("No changes made to registry.")
        Exit Sub
    End If
    Dim reg As RegistryKey
    reg = Registry.ClassesRoot.CreateSubKey( _
        name & ".Connect\CLSID")
    reg.SetValue(Nothing, guid)

    reg = Registry.LocalMachine.CreateSubKey( _
        "SOFTWARE\Microsoft\VisualStudio\7.0\AddIns\" & _
        name & ".Connect")
    reg.SetValue("FriendlyName", friendlyname)
    reg.SetValue("Description", description)
    reg.SetValue("LoadBehavior", 4)
    reg.SetValue("CommandLineSafe", 1)
    reg.SetValue("CommandPreload", 1)
    MsgBox("Finished!")
End Sub

```

This macro uses the Registry class, which is in the Microsoft.Win32 namespace of the .NET framework. Using the name, GUID, and description, it enters the correct data into the Registry so that the next time you start Visual Studio .NET, the IDE will know about your macro.



If you use the preceding macro, you need to enter the GUID. This can be somewhat difficult, so I recommend that before running it, you use the mouse to highlight the GUID in the add-in code and copy it to the clipboard. Then when you run the macro and you're asked for the GUID, you can just paste it in.

Creating an Add-in

In this section I walk you through the creation of an add-in. First I walk you through the use of the wizard, then I show you how to build your add-in. Along the way, I give you tips for dealing with problems you might encounter.



When you run the Add-in wizard, it will create two projects: the add-in project, and a Setup (also called Deployment) project for the add-in. However, initially, the wizard sets up the solution configuration such that the Setup project is set to not build when you build the entire solution. When you build the entire solution, you will see a message at the end such as this:

Build: 1 succeeded, 0 failed, 1 skipped

The “1 skipped” causes a lot of confusion. But getting Visual Studio .NET to not skip the project is simple. You can either right-click the Setup project and choose Build to force a build on it, or you can set it to always build, by right-clicking on the solution name in the Solution Explorer and then, in the popup menu, by choosing Properties. In the Properties window, click on Configuration Properties. In the list on the right, check the box under Build for the Setup project; then click OK. Now your project is set to build each time.

Using the Wizard

To create an add-in, follow these steps:

1. From the main IDE, open the New Project dialog box (either through File⇨New Project or File⇨Add Project).
2. In the New Project dialog box, in the Project Type treeview, choose Other Projects⇨Extensibility Projects.
3. In the Templates list, choose Visual Studio .NET Add-in.
4. As with any project, enter a name and location for the project.
5. Click OK.

After you click OK, the Visual Studio Add-in wizard will begin. The first page of the wizard is simply an introductory message; just click Next. (Or, if you realize you want to change your settings in the New Project dialog box, go back by clicking Back.) Then follow these steps:

1. On page 1 of 6 (note that the introductory page doesn't have a page number), choose the language you want to use for your add-in. Remember, unlike macros, you can use any .NET language that's installed on your computer. For the Enterprise Edition of Visual Studio .NET, your choices are Visual C#, Visual Basic, and Visual C++/ATL. For these sample steps, I'm choosing C#. Then click Next.
2. On page 2 of 6, choose in which IDE you want the add-in to be used: the main IDE or the Macros IDE or both. The first checkbox on this page is Microsoft VSMacros IDE, which, in this book, is what I've been calling the Macros IDE. The second checkbox is Microsoft Visual Studio .NET, which is what I've been calling the main IDE. For this example, check just the main IDE. Then click Next.
3. On page 3 of 6, specify the name and description of your add-in. Then click Next.



On page 3 of 6, the phrase “name of your Add-in” refers to the friendly name, that is, the name that will appear in the upper list of the Add-in Manager. This is separate from the name by which the IDE actually knows the add-in.

4. On page 4 of 6, you are offered four checkbox options:
 - The first option asks whether you want to create a menu item for the Tools menu after you've created and installed the add-in. (Yes, that's what this checkbox option really means, even though the wording is particularly bad.) That said, it also offers another, more secret option: whether or not your class will be derived from `IDTCommandTarget`. I pretty much always want my classes derived from `IDTCommandTarget`, so I always check this first option.
 - The second option asks whether you want to use the add-in with the command-line version of the Visual Studio .NET. If you do, your add-in is not allowed to open up a modal dialog box. By checking the second checkbox, you're agreeing not to put up any modal dialog boxes.
 - The third option asks whether you want the add-in to load when the main IDE loads (after the add-in is installed, of course).
 - The final option asks whether you want your add-in to be available to all the users on the present machine, or just the current user (i.e., you).
5. For this example, leave the middle two checkboxes checked, and check the final box to make the add-in available to all users. Then click Next.



You can change the options on page 4 later on if you prefer. To remove the item from the Tools menu, use Tools⇨Configure, the way you would to normally remove a menu item. To change whether to use the add-in in command-line mode, or to change whether the add-in loads when the IDE loads, use the Add-in Manager, as I describe later in this chapter, in "Managing Add-ins." To change whether the add-in is available to just you or to all users, open regedit and manually move the add-in information from HKEY_LOCAL_MACHINE to HKEY_CURRENT_USER, or vice versa. A word of caution here: Making such moves is often risky, due to the possibility of making mistakes, so be extremely careful if you decide to do this.

6. On page 5 of 6, you can specify information in the About box, specifically, whether you want a description of your add-in to appear in the IDE's About box. On this page you will also see an icon appear automatically, indicating that you can change the icon later. Click Next.
7. On page 6 of 6, you will see a list of your chosen options for your review. If you want to change any of them, click Back. When you're finished, click Finish.



Remember, your add-in gets registered with Visual Studio .NET when you are finished running this wizard. It's during this registration that Visual Studio .NET makes changes to the Registry, which is the process by which Visual Studio .NET registers your add-in. In order to make the add-in available to all the users, Visual Studio .NET will register the add-in under the

HKEY_LOCAL_MACHINE key. If the add-in is just for you, Visual Studio .NET will register it under the **HKEY_CURRENT_USER** key. Under either of these two keys, Visual Studio .NET will store the add-in information in **HKEY_LOCAL_MACHINESOFTWARE\Microsoft\VisualStudio\7.0\AddIns**. (See “Managing Add-ins,” later in this chapter for more information on the Registry.) However, remember, if you change these settings, you’re only changing the present installation; you won’t be changing the installer application.

After you click Finish, Visual Studio .NET will create *two* projects for you: the add-in project and a Setup program for your add-in. The Setup program is what you give other IDE users so they can install your add-in. Visual Studio .NET will also automatically set up a dependency between the two projects: The Setup project depends on the add-in project; thus, if you build the Setup project, the add-in project will build as well.



If you go looking for the Setup project on your hard drive, be aware that the wizard creates it under the directory for your add-in. If, for example, you named your add-in MyAddin1, the wizard would create a directory called MyAddin1 to contain your add-in project, and then a directory called MyAddin1\MyAddin1Setup. When you build the Setup, you will get a directory MyAddin1\MyAddin1Setup\Debug or MyAddin1\MyAddin1Setup\Release, which contains the debug and release version of the Setup file, respectively. Finally, the Setup file is called MyAddin1Setup.msi. (These days, Microsoft has its own installation software that’s automatically installed with the different breeds of Windows, and the setup information goes inside the .msi file.)



If, on page 4, you specified that you want the add-in available only to you, not all users, this information will carry forward into the Setup program. When the IDE user who receives your Setup program installs your add-in, the add-in will install only for that IDE user, not other users of the same computer.



Important: If you see an error message such as this in your task list:

```
C:\dev\MyAddin\MyAddinSetup\MyAddinSetup.vdproj Unable to find
dependency 'C:\dev\MyAddin\obj\Debug\MyAddin.tlb' of project
output 'Primary output from MyAddin (Active)'
```

fear not. This error refers to the Setup project, not the add-in itself, meaning you can still run the add-in. This error is simply a side effect from the creation by the wizard of the add-in’s Setup project before building the add-in project.

The Setup project is looking for the .tlb file (which contains type information about the add-in for the COM system), but the .tlb file is not there, and won't be until you build the add-in. To make the message go away, however, there's a slight catch: Even after you build the add-in, the message will still be in the task list. To get rid of it, right-click the Setup project, and in the Popup menu choose Build.

The following is the code that was generated automatically for the add-in I created. For this particular add-in, I chose to have a Tools menu item automatically created for me and to have the project written in C#. (Note that the wizard also automatically inserted several comments, but to save space here, I removed these comments from the code.)

```
namespace CSAddin1
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;
    using System.Runtime.InteropServices;
    using EnvDTE;

    [GuidAttribute("2BD7B62D-1595-4D4E-93C6-02263436DEB8"),
     ProgId("CSAddin1.Connect")]
    public class Connect : Object, Extensibility.IDTExtensibility2,
        IDTCommandTarget
    {
        public Connect()
        {
        }

        public void OnConnection(object application,
            Extensibility.ext_ConnectMode connectMode, object addInInst,
            ref System.Array custom)
        {
            System.Windows.Forms.MessageBox.Show("Add-in loaded!");
            applicationObject = (_DTE)application;
            addInInstance = (AddIn)addInInst;
            if(connectMode == Extensibility.ext_ConnectMode.ext_cm_UISetup)
            {
                object []contextGUIDS = new object[] { };
                Commands commands = applicationObject.Commands;
                _CommandBars commandBars = applicationObject.CommandBars;

                try
                {
                    Command command = commands.AddNamedCommand(addInInstance,
                        "CSAddin1", "CSAddin1", "Executes the command for CSAddin1",
                        true, 59, ref contextGUIDS,
                        (int)vsCommandStatus.vsCommandStatusSupported
                        +(int)vsCommandStatus.vsCommandStatusEnabled);
                }
            }
        }
    }
}
```



```
        CommandBar commandBar = (CommandBar)commandBars["Tools"];
        CommandBarControl commandBarControl =
            command.AddControl(commandBar, 1);
    }
    catch(System.Exception /*e*/)
    {
    }
}

}

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

public void OnAddInsUpdate(ref System.Array custom)
{
}

public void OnStartupComplete(ref System.Array custom)
{
}

public void OnBeginShutdown(ref System.Array custom)
{
}

public void QueryStatus(string commandName,
    EnvDTE.vsCommandStatusTextWanted neededText,
    ref EnvDTE.vsCommandStatus status, ref object commandText)
{
    if(neededText ==
        EnvDTE.vsCommandStatusTextWanted.vsCommandStatusTextWantedNone)
    {
        if(commandName == "CSAddin1.Connect.CSAddin1")
        {
            status =
                (vsCommandStatus)vsCommandStatus.vsCommandStatusSupported|
                vsCommandStatus.vsCommandStatusEnabled;
        }
    }
}

public void Exec(string commandName, EnvDTE.vsCommandExecOption
    executeOption, ref object varIn, ref object varOut,
    ref bool handled)
{
    handled = false;
}
```

```

if(executeOption ==
EnvDTE.vsCommandExecOption.vsCommandExecOptionDoDefault)
{
    if(commandName == "CSAddin1.Connect.CSAddin1")
    {
        handled = true;
        return;
    }
}
private _DTE applicationObject;
private AddIn addInInstance;

}
}

```

Other than removing the comments, the only other change I made to this code was to add a call to `MessageBox` in the beginning of the `OnConnection` function:

```
System.Windows.Forms.MessageBox.Show("Add-in loaded!");
```

But in order for the program to recognize the `System.Windows.Forms` namespace, I also had to add a reference. To do this, I right-clicked the word `References` in the Solution Explorer under my project name. Using the references dialog box, I added a reference to `System.Windows.Forms.dll`.

If you want to add this line for a VB.NET add-in, here's the line you will type instead in the `Connect.VB.NET` source file (Visual Basic has its own `MsgBox` function, and so you don't need to add a reference to the `System.Windows.Forms` namespace as I did with the C# version):

```
MsgBox("Add-in Loaded!")
```

And, finally, if you want to add this line to the C++ version, you can just call the Win32 API function `MessageBox`, like so:

```
MessageBox(0, "Add-in Loaded!", "Add-in", MB_OK);
```

You can see that the two interfaces I described in the previous section are indeed covered. The class includes the functions `OnConnection`, `OnDisconnection`, `OnAddInsUpdate`, `OnStartupComplete`, and `OnBeginShutdown`. All five of these functions are for the `IDTExtensibility2` interface. The class also has two functions, `QueryStatus` and `Exec`, which are part of the `IDTCommandTarget` interface. And if you look at the class header itself, you can see that your class is derived from the two interfaces:

```
public class Connect : Object, Extensibility.IDTExtensibility2,
IDTCommandTarget
```

Building and Running the Add-in

To build the add-in, simply right-click the add-in's name in the Solution Explorer, and in the popup menu choose Build, just as you would with any other project. When you build the add-in, the build process will make sure the Registry is set up for the COM component that gets built. However, as I mentioned in the section "Add-ins Are COM Components," the build process registers the COM component only with the COM system; the process does not register the add-in component with the IDE. (In the same section, I showed you how to register the component manually and provided a macro to help in the effort.)



When you build an add-in, be aware of these points. First, remember that the add-in runs as an enhancement to the very same IDE that you are using to build the add-in. That means if the add-in is running in the current instance of the IDE or in another instance of the IDE, you will not be able to link the object codes into a single DLL. The IDE will have the file open and the operating system will put a lock on it. Therefore, before you can link the object files, make sure you do not have the add-in currently running.

The wizard you originally used to create the add-in project automatically configured the project, so that when you run the wizard, the IDE will launch a second instance of Visual Studio .NET, in which your add-in will run. There's nothing special about this newly running instance of Visual Studio .NET; it's just another instance running, with no command-line parameters and no special provisions for your add-in. Therefore, to try out your add-in, set the add-in project as the startup project; press F5, to run the add-in in debug mode, or Ctrl-F5, to run it without debugging.

If, however, instead of the second instance of Visual Studio .NET starting, you get a message stating that no program has been specified (or, in the case of C++, a message box appears asking for the name of the executable file (you need to configure the project to run Visual Studio .NET. The following points tell you how, depending on the language of the project.

- If you write your add-in in C#, open the properties for the project; in the Property Pages dialog box in the left treeview, choose Configuration Properties → Debugging. Then in the properties on the right, under Start Action, set Start Application to the full path of your Visual Studio .NET application.
- If you write your add-in in VB.NET, open the properties for the project; in the Property Pages dialog box in the left treeview, choose Configuration Properties → Debugging. In the properties on the right, under Start Action, set Start External Program to the full path to Visual Studio .NET.
- If you write your add-in in C++, you can simply fill in the path to the Visual Studio .NET program right in the executable section of the dialog box that appears when you try to run your add-in. Filling it in here will set the correct project property for you automatically. Alternatively, you can manually set the

project property. To do so, open the properties for the project, and in the left treeview of the Property Pages dialog box, choose Configuration Properties→Debugging. Then in the properties on the right, under the heading Action, set the Command property to the full path to your Visual Studio .NET program. (Remember, you'll probably want to do all this for both the Debug and Release configurations.)

In all three cases, the executable is devenv.exe, and by default the program is in C:\Program Files\Microsoft Visual Studio .NET\Common7\IDE\devenv.exe.

Managing Add-ins

To control the loading and unloading of the add-ins, the IDE user can invoke the Add-in Manager, which lists all the add-ins that are registered with the IDE. To access the Add-in Manager, choose Tools→Add-in Manager. When it opens, you will see a list of the add-ins that is registered with Visual Studio .NET. (These are the add-ins listed under the HKEY_LOCAL_MACHINESOFTWARE\Microsoft\VisualStudio\7.0\AddIns key in the Registry.) Each add-in has a checkbox to its left, as well as checkboxes listed under the two columns labeled Startup and Command Line. Here's what these checkboxes do:

- If you check the leftmost box, the add-in will load as soon as you exit from the Add-in Manager. If the add-in is already loaded and you uncheck the checkbox, the add-in will unload when you exit the Add-in Manager.
- If you check the box under Startup, you will specify that the add-in will load at startup the next time Visual Studio .NET starts. Clearing this means the add-in will not load automatically at startup.
- If you check the box under Command Line, you will specify that the add-in is safe to be loaded when Visual Studio .NET is executed as a command line.

In addition to using the Add-in Manager, you can access more options by directly editing the Registry. In general, this is not a good idea; however, the Add-in Manager does not cover all the options for the add-ins. If you start regedit and go to the HKEY_LOCAL_MACHINESOFTWARE\Microsoft\VisualStudio\7.0\AddIns key in the Registry, you will see the names of the add-ins. If you then click on one of the names, in the right panel of the regedit, you will see some or all of the following values:

CommandPreload. This is a handy value that your add-in can check to determine if the add-in has been run since it was first installed: 0 means the add-in has been run, 1 means it has not. If the value is 1, then your OnConnection function will receive the value ext_cm_UISetup for the ConnectMode parameter.

Description. This is the description that appears in the bottom portion of the Add-in Manager when you select an add-in.

FriendlyName. This is the name that appears in the Add-in Manager. If you don't supply a value here, the name will be the actual name of the add-in.

LoadBehavior. This specifies the current load situation for the add-in. The value will be a sum of the following: 1 means the add-in should be loaded when the IDE starts; 2 means the add-in is presently loaded; and 4 means the add-in should be loaded when Visual Studio .NET is invoked as a command-line utility for building projects. (The 1 corresponds to a check under Startup in the Add-in Manager, and 4 corresponds to a check under Command Line in the Add-in Manager.)

SatelliteDllName. This is the name of a DLL that contains resources such as icons and string resources that your add-in can use. (The main reason for separating the string resources into a satellite DLL is so you can include strings in different languages. You would have separate resource directories for each *locale*. If you're familiar with working with locales, you follow the standard procedure for this.)

SatelliteDllPath. This is the path that contains the satellite DLL. However, the IDE constructs the actual path by taking this path and appending a number representing the locale.

AboutBoxDetails. When you choose Help⇨About Microsoft Development Environment, you will see a list of the languages installed in the Visual Studio .NET, as well as a list of the add-ins that are installed *and running*. Below the list is a box with an icon and a description. By setting the AboutBoxDetails key, you can specify a string that appears as the description in this box. (This is separate from the Description value.)

AboutBoxIcon. In addition to a description in the About dialog box, you can specify an icon that the About dialog box should display when the user clicks on the name of your add-in. To specify an icon, use an integer that is the icon number in the SatelliteDll.



Since you can also create an add-in for the Macros IDE, you will find the same Registry setup as in the preceding list, under the key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VSA\7.0\AddIns`. This key lists the add-ins that apply to the Macros IDE.

Moving Forward

This chapter covered the add-in concepts. In the two chapters that follow, I take these concepts and build several add-ins. During the process you will see just how easy it is to build a powerful add-in that's fully integrated to the Visual Studio .NET IDE.

Creating Add-ins for the IDE

In this chapter, I expand on the concepts presented in Chapter 6 to show you how you can build an add-in that includes a GUI and interacts with the IDE's command system and menu bar.



In the code for each of the sample add-ins in this chapter, you will notice a GUID value in the form of a GuidAttribute. These are the GUIDs that were assigned to me when I ran the wizard. Although you can certainly use the same GUID in your code if you decide to type in these samples, I don't recommend doing so. The reason is that if you create a product that uses the GUID, and by chance another reader does the same thing, and both your add-ins end up on the same computer, there will be a problem. Therefore, I recommend that you run the Add-in wizard to get the project started and use the GUID that the wizard assigns to you. From there, you can type in the code from the samples.

Building an Add-in in C++

If you built your add-in in C++, you probably quickly noticed something disturbing: The add-in does not use .NET and the C++ managed extensions by default! That's not good. Fortunately, you have a choice: You can either add managed extensions or you

can continue writing the add-in without the .NET framework, accessing the automation objects through the COM objects that are passed into the functions in your add-in's main `CConnect` class. (Yes, in C++, the class has a C in front of it.)

If you want to skip the .NET framework and the C++ managed extensions, you can write your add-in, continuing from the starter code the wizard generated for you. But if you want access to the .NET framework and the C++ managed extensions, here are the steps you can follow to add it.



If these steps seem a little strange, it's because the compiler option for managed extensions is not compatible with several other compiler options.

1. Open the project properties.
2. In the treeview, choose Configuration Properties→General. In the properties on the right, set Use Managed Extensions to Yes.
3. Choose Configuration Properties→C/C++→Code Generation. Set the Use Runtime Checks property to Default. Also in this section set the Enable Minimal Build property to No.
4. Choose Configuration Properties→C/C++→General. Make sure Debug Information Format is set to Program Database (/Zi).

After you follow these steps, your project will be set up to build a .NET assembly rather than a standard DLL, which means you'll be able to access the .NET framework classes.



If you're planning to use C++ to write your add-in, you will want to have a thorough understanding of COM and ATL. The alternative is to either learn about COM and ATL (perhaps pick up a book on it) or seriously consider writing your add-in in C# or VB.NET. (If you prefer C++ style syntax, then try C#. Its syntax is very much like C++, and it really is a good language.)

Including a GUI with Your Add-in

If you do not expect your add-in to be run in command-line mode, you can include a GUI. There are two different ways your add-in can have a GUI:

Interact with the IDE directly. This approach is useful for creating, for example, additional tool windows that are integrated right with the IDE.

Use the `System.Windows.Forms` library to build a form yourself. This is best if you need to display a modal dialog. (I don't recommend using this approach for nonmodal windows, as the windows will not be integrated to the IDE.)

In the sections that follow I'll show you how to use the `Windows.System.Forms` library; then I demonstrate how to create a tool window using your add-in. In the case of the tool window, I cover two aspects: putting a preexisting ActiveX control in the tool window, and putting a form that you create with the IDE's Designer inside the tool window.

Working with Commands

You can access most of the functionality of the IDE through named commands, such as `Edit.SelectAll`, which is the command for selecting all the text in the currently open source code document. You can set up your add-in to recognize named commands. In the sections that follow, I will be showing you how to do so; but before we get started, I need to point out that for your add-ins to use named commands, the class for the add-in must be derived not only from `IDTExtensibility2`, but also from `IDTCommandTarget`. The `IDTCommandTarget` interface provides the two functions that are necessary for working with commands, `QueryStatus` and `Exec`.



If you need to remove a command that you added (as you'll do in the examples in the following section), write a macro or add-in code that obtains the reference to the `Command` object for the command, and call the `Delete` member function. (Don't worry about accidentally deleting a built-in command. I tried this on my computer—choosing a command I figured I could live without in the event of the worst-case-scenario—and I received an error; fortunately, Visual Studio .NET wouldn't let me delete it.) Here's an example of a macro that deletes a command created by one of the examples later in this chapter (in the section, "Building a Tool Window").

```
Sub DeleteCommand()
    Dim cmd As Command =
DTE.Commands.Item("ToolWindowAddin.Connect.Load")
    If Not cmd Is Nothing Then
        cmd.Delete()
    End If
End Sub
```

Using the Forms Library

For this sample, I started by using the Add-in wizard to create a C# add-in. Since I wanted my add-in to support commands, I chose the Create Tools menu option in the wizard. From there I added code so that when the add-in receives a command, it will display a model dialog box, which I hard-coded in C#. (You can also use the Designer to create your form, in which case, in the following code, instead of hard-coding the form information, just call `ShowDialog` for your form.)

The beginning of the namespace looks like this (make sure you add a reference in your project to System.Windows.Forms.dll):

```
namespace GuiAddin1
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;
    using System.Runtime.InteropServices;
    using EnvDTE;

    // For this next line, be sure to add a reference
    // in your project to System.Windows.Forms.dll.
    using System.Windows.Forms;
```

Next comes the class header and the `OnConnection` method. Remember, I checked the option `Create Tools` menu item in order to have my class implement both `IDTExtensibility2` and `IDTCommandTarget`. However, I didn't really want the extra goodies the `Tools` menu option adds—a toolbar and such—so I deleted those lines, but kept the part where the code creates a named command (although I fixed up the call to `AddNamedCommand` with my own command name):

```
[GuidAttribute("3B357420-B183-4553-9851-D70B7C60BD19"),
ProgId("GuiAddin1.Connect")]
public class Connect : Object, Extensibility.IDTExtensibility2,
    IDTCommandTarget
{
    public Connect()
    {
    }

    // Note: Removed Toolbar stuff
    // Removed connectMode comparison. Was:
    // if(connectMode ==
Extensibility.ext_ConnectMode.ext_cm_UISetup)
    public void OnConnection(object application,
        Extensibility.ext_ConnectMode connectMode,
        object addInInst, ref System.Array custom)
    {
        applicationObject = (_DTE)application;
        addInInstance = (AddIn)addInInst;
        object []contextGUIDS = new object[] { };
        Commands commands = applicationObject.Commands;

        try
        {
            Command command =
commands.AddNamedCommand(addInInstance,
                "MyCommand1", "GUIAddin1 MyCommand1",
```

```

        "Executes the command for GuiAddin1.MyCommand1",
        true, 59, ref contextGUIDS,
        (int)vsCommandStatus.vsCommandStatusSupported +
        (int)vsCommandStatus.vsCommandStatusEnabled);
    }
    catch(System.Exception )
    {
        MessageBox.Show("Exception...");
    }
}

```

I didn't add any code to the `OnDisconnection`, `OnAddInsUpdate`, `OnStartupComplete`, and `OnBeginShutdown` methods. However, you can't delete them, because if you do, you'll have an abstract class that you cannot instantiate. Therefore, leave them as-is.

Next I fixed up the `QueryStatus` method just a bit so it understands my own command. (Remember, the `QueryStatus` function is what the IDE calls to find out if a command is available, and the `QueryStatus` function will receive the fully qualified name of the command. Therefore, if you named your add-in something other than what I did—`GuiAddin1`—then you'll have to replace the text `GuiAddin1` in this code with your add-in name. Here's the code:

```

public void QueryStatus(string commandName,
    EnvDTE.vsCommandStatusTextWanted neededText,
    ref EnvDTE.vsCommandStatus status, ref object commandText)
{
    if(neededText ==
        vsCommandStatusTextWanted.vsCommandStatusTextWantedNone)
    {
        if(commandName == "GuiAddin1.Connect.MyCommand1")
        {
            status =
                vsCommandStatus.vsCommandStatusSupported |
                vsCommandStatus.vsCommandStatusEnabled;
        }
    }
}

```

Finally, the `Exec` method is where I create the form. The idea is that when the IDE user issues a command to the add-in, the add-in will respond by opening the form. You can see in this code that I created a new `Form` instance and set various properties. Then I created several control instances and set their properties. After I created the control instances, I set the `Form` instance's `AcceptButton` property equal to my OK button so that when the IDE user presses `Enter`, the form will close as if he or she pressed `OK`. Here's the code, which finishes the class:

```

public void Exec(string commandName,
    EnvDTE.vsCommandExecOption executeOption,
    ref object varIn, ref object varOut, ref bool handled)

```

```

    {
        handled = false;
        if(executeOption ==
vsCommandExecOption.vsCommandExecOptionDoDefault)
        {
            if(commandName == "GuiAddin1.Connect.MyCommand1")
            {
                Form f = new Form();
                f.Height = 180;
                f.Width = 200;
                f.Text = "GUI Add-in";
                Label label1 = new Label();
                label1.Text = "Enter your name";
                label1.Left = 20;
                label1.Top = 20;
                TextBox tb1 = new TextBox();
                tb1.Text = "";
                tb1.Left = 20;
                tb1.Top = 60;
                Button okbtn = new Button();
                okbtn.Text = "OK";
                okbtn.Left = 20;
                okbtn.Top = 120;
                okbtn.DialogResult = DialogResult.OK;
                Button cancelbtn = new Button();
                cancelbtn.Text = "Cancel";
                cancelbtn.Left = 100;
                cancelbtn.Top = 120;
                cancelbtn.DialogResult = DialogResult.Cancel;
                f.Controls.AddRange(new Control[]
                    {label1, tb1, okbtn, cancelbtn} );
                f.AcceptButton = okbtn;
                DialogResult res = f.ShowDialog();
                if (res == DialogResult.OK)
                {
                    MessageBox.Show(tb1.Text);
                }
                handled = true; // Remember to set this to true!!!
            }
        }
    }

private _DTE applicationObject;
private AddIn addInInstance;

}
}

```

When you compile and run this code, you have an add-in that can display a dialog box with the issuance of a command. Here's how to do that:



Figure 7.1 The add-in will display a modal dialog box.

1. Run a new instance of Visual Studio .NET.
2. Load your add-in using the Add-in Manager.
3. Choose View→Other Windows→Command Window to open the command window.
4. In the command window, type the following command and press Enter (although if you named your add-in something other than `GuiAddin1`, then you'll replace `GuiAddin1` with the name of your add-in).

```
GuiAddin1.Connect.MyCommand1
```

When you run this command, the IDE will first call your add-in's `QueryStatus` function, passing it the exact string you typed, `GuiAddin1.Connect.MyCommand1`. Your `QueryStatus` function will indicate the command is available by setting the status parameter. Next, the IDE will call your `Exec` function, asking you to execute the command. The `Exec` function will display a modal dialog box, as shown in Figure 7.1, allowing you, in this case, to type in a name. When you press Enter, the `Exec` function will open a message box showing the string you typed in. Although the message box in itself is not particularly interesting, the code for this message box demonstrates how you can retrieve user information from the form by inspecting the properties of the controls on the form.

Building a Tool Window

Before I delve into the code for creating a tool window, I want to explain the role of ActiveX controls and tool windows. In short, tool windows are *not* .NET windows, hence they can only display ActiveX controls, not .NET controls.

All the tool windows in the IDE are implemented as ActiveX controls. You can see this for yourself if you open up the `regedit`. Inside `regedit`, drill down to `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.0\ToolWindows`. Under the `ToolWindows` key, you will see several GUIDs. These are not yet the GUIDs for ActiveX controls; these are unique identifiers for the tool windows. Every tool window has to have a unique identifier, so what better way than to use GUIDs? If you click on this GUID, `{3AE79031-E1BC-11D0-8F78-00A0C9110057}`, you will see two values for it: a name and another GUID. The name is `Solution Explorer`, so this tool window represents the `Solution Explorer`. The GUID is `{DA9FB551-C724-11d0-AE1F-00A0C90FFFC3}`. If you

dig through the `HKEY_CLASSES_ROOT/CLSID` key, you'll eventually find this GUID, which is for an ActiveX control.

In order for your add-in to create a tool window, you need two identifiers: a GUID that will identify the tool window and a unique identifier for the ActiveX control. For the ActiveX control you can either specify the ActiveX control's own GUID or the official name for the ActiveX control. For the sample that follows, I used the built-in Internet Explorer control, and its unique name is `Shell.Explorer`.



If you're not an ActiveX control expert and all this seems a bit daunting, don't worry. In the next section I show you how you can get around the fact that the tool window needs an ActiveX control. Nevertheless, you still might want to review this section so you can fully understand how tool windows work.

Once you have the two unique identifiers, you can call `CreateToolWindow`, passing a reference to your add-in (so that the IDE will know which add-in goes with this new tool window), the unique name (or GUID) for the ActiveX control, a title for the tool window, and a unique GUID for the tool window. Finally, as a fifth parameter you specify a reference to an object that will receive a *document object* returned by `CreateToolWindow`. This document object is, in fact, the actual object that you'll use to communicate with the ActiveX control. You'll see how to do that in the sample code that follows.

This sample code creates a tool window that displays a Web browser; specifically, it displays an instance of the browser control that is the heart of Internet Explorer. (In the past I would have instructed you to make sure you had IE installed on your computer, but these days it's there automatically.) I chose the Web browser control, for two reasons. First, I wanted to use a control that's easy to use; and second, I wanted to use a control that I know you'll have installed on your computer.

To set up this sample, create a new C# add-in, call it `ToolWindowAddin`, and check the wizard box that asks whether you want a tool menu item created. Next—and this is easy to forget—you *must* add a reference to the ActiveX control that your tool window will be hosting. When you right-click the project name in the Solution Explorer and choose Add Reference, you'll see a tab called COM in the Add Reference dialog box; click on it and wait a moment as the COM list loads. (This could take awhile depending on how much software you have installed on your computer.) Scroll down, find, then double-click Microsoft Internet Controls (its path is `C:\Windows\System32\shdocvw.dll`). Then click OK.

When you click OK, two things will happen. First, believe it or not, the IDE will automatically create an assembly based on the ActiveX control. It will name this assembly `Interop.SHDocVw.dll` (the middle name, `SHDocVw`, is the name of the library) and place it in the `obj` directory underneath your project directory. Second, the IDE will add to your project a reference to this new assembly.



When you add the Microsoft Internet Controls component, the IDE will also automatically add the control to the Setup project for you. You don't want that, however, because the Setup project will refuse to build with a Windows system file present. To fix this, go to the Solution Explorer and find the Setup

project called ToolWindowAddinSetup. Expand the project folder and then expand the Detected Dependencies folder. Under Detected Dependencies, right-click on shdocvw.dll and in the popup menu choose Exclude.

Now in your code you can access the types specified in the ActiveX control. The Web browser's type is `IWebBrowser`. So when I call `CreateToolWindow`, the final parameter, which receives a reference to an object, will receive a reference to `IWebBrowser`.

The `IWebBrowser` control has a simple method called `Navigate`. You pass a URL, and the control will load and display the URL.

Following is the first part to the add-in. You can see that I added a reference to `System.Windows.Forms`, which I did simply to give me access to the `MessageBox` function.

```
namespace ToolWindowAddin
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;
    using System.Runtime.InteropServices;
    using EnvDTE;

    // Remember to reference System.Windows.Forms in Solution Explorer
    using System.Windows.Forms;
}
```

Next is the `Connect` class that the wizard generated automatically for me. In this class you can see that I added a couple of member variables, one called `doc` (which is the `IWebBrowser` instance) and one called `toolwin` (which is an instance of `Window` representing the tool window). Later on in this code, I needed a GUID, so I ran the `Create GUID` tool (available through the IDE by choosing `Tools` → `Create GUID` or by running `guidgen.exe` from the command line). I then pasted the GUID in as the value for a string called `newguid`.

After that, I created a variable called `tempdoc`, which will temporarily hold the value of the object returned by `CreateToolWindow`. The reason I needed this temporary variable is that `CreateToolWindow` insists that I pass to it a variable of type `object`. Then after calling `CreateToolWindow`, I cast the result to my `IWebBrowser` variable.

The call to `CreateToolWindow` also returns an instance of `ToolWindow`. (This instance is the actual return value; `CreateToolWindow` returns my Web browser document through a reference parameter.) I save the `ToolWindow` instance so I can use it later, and then I set its `Visible` property to `True`.

Then, just for fun (and to prove to myself that this code really worked), I call the Web browser control's `Navigate` function, passing a URL. But the `Navigate` function requires four extra parameters that aren't really needed at this point, so I pass four empty objects.

Finally, all this code that I just described goes in a `try` block, which I used for debugging as I wrote the code. I left the `try` block there, in case I want to add any more error checking. For instance, if for some reason `CreateToolWindow` is unable to load the ActiveX control, the CLR will throw an exception, which the code can catch. But if the code doesn't catch the exception, the IDE will display an error message when the IDE

user tries to load the add-in, asking if he or she would like to unregister the add-in. (Best to handle such errors yourself, where you have more control. You might, for example, display a `MessageBox` explaining that the ActiveX control is not available, and offer suggestions on how the IDE user can obtain it.)

Here is the part of the code I just described:

```
[GuidAttribute("0FA19594-9EAC-4215-8F05-345D8BC4A10C"),
ProgId("ToolWindowAddin.Connect")]
public class Connect : Object, Extensibility.IDTExtensibility2,
    IDTCommandTarget
{

    private SHDocVw.IWebBrowser doc = null;
    private Window toolwin = null;

    public Connect()
    {
    }

    public void OnConnection(object application,
        Extensibility.ext_ConnectMode connectMode,
        object addInInst, ref System.Array custom)
    {
        applicationObject = (_DTE)application;
        addInInstance = (AddIn)addInInst;

        try
        {
            String newguid = "{8F4C6171-1167-4d64-82B7-A25F99A29E0A}";
            object tempdoc = null;
            Window macroexp = applicationObject.Windows.Item(
                Constants.vsWindowKindMacroExplorer);
            String objkind = macroexp.ObjectKind;

            toolwin = applicationObject.Windows.CreateToolWindow(
                addInInstance,
                "Shell.Explorer",
                "New Tool Window", newguid, ref tempdoc);
            toolwin.Visible = true;
            doc = (SHDocVw.IWebBrowser)tempdoc;
            object emptyobj = null;
            doc.Navigate("http://www.wiley.com", ref emptyobj,
                ref emptyobj, ref emptyobj, ref emptyobj);

        }
        catch (System.Exception e)
        {
            MessageBox.Show(e.Message);
        }
    }
}
```

In the next section of code I register the commands. This add-in has two commands, one that shows the tool window and one that loads a Web address into the tool window. Notice in the first call to `AddNamedCommand`, for the third parameter, instead of describing the command, I simply passed a string with the value `"Toolwin"`. The reason is that I'm adding a menu item on the View window, inside the Other Windows submenu. The name that I want to appear there is the name of my window, and that name will come from the description parameter of the call to `AddNamedCommand`. That way my window will appear in the Other Windows list along with windows such as Macro Explorer; these other windows also have just a window name, not a full description.

After the code for the creation of the two commands comes the code that creates a menu item. Pay close attention to this code, as it took a good deal of digging on my part. The menu item objects are actually part of the Microsoft Office toolkit, not part of .NET. Thus, the Visual Studio .NET documentation's coverage of the menu objects is sparse at best. (In my digging I ended up at Microsoft's MSDN Library site, <http://msdn.Microsoft.com/library>.)

The menu modification code is a bit convoluted. The menu of items across the top of the main IDE window is considered a `CommandBar` instance. But this instance only has the names across the top, not the names on the drop-down menus. Further, each name across the top is an instance of `CommandBarPopup`. (Normally in a `CommandBar`, the items are instances of `CommandBarCommand`, but in the case of menu items, they are instances of `CommandBarPopup`, which is derived from `CommandBarCommand`.)

Each drop-down menu is itself an instance of `CommandBar`. To obtain this, I ask the `CommandBarPopup` that corresponds to the View menu for its `CommandBar` instance. That gives me the `CommandBar` instance of the View menu.

From there I do it all over again: I find the `CommandBarPopup` member corresponding to "Other Windows", and grab its `CommandBar` instance (which "Other Windows" has, since it's a submenu.) And that final `CommandBar` instance is the instance where I want to insert my command. (*Whew!*)

Finally, to add the command, I call a member function of the `Command` instance, not a member of the `CommandBar` instance. Specifically, I call `AddControl`, passing the `CommandBar` instance, and a number representing the position where I want to insert the command. In my call to `AddControl` in the following code you can see that I passed the variable called `othersBar` and the number 1. Thus, after this code runs, I'll have a menu item `View⇄Others Windows⇄ToolWin`.

One more point about this code: When you add a set of commands to the IDE and a menu to the IDE, it's almost a done deal. The commands and the menu won't go away when the add-in unloads. And when you restart the IDE, the commands and menu will still be there, even if the add-in isn't loaded yet. And what if you reload the add-in? Not a problem. Notice there's a `try/catch` block around this code. If the `AddNamedCommand` function discovers that the command already exists, the function will throw an exception, sending me down to my catch block, skipping my attempt to add a menu item. And so I will still only have one menu item, rather than an additional one each time the add-in loads. (Therefore, and contrary to what the default code from the wizard does, I do not need a big if statement around this code.)


```
object []contextGUIDS = new object[] { };
Commands commands = applicationObject.Commands;

try
{
    // If the commands already exist, this whole block will
    // drop out to the catch block below, so I don't need
    // to worry about accidentally adding anything twice.
    Command command1 = commands.AddNamedCommand(addInInstance,
        "Show", "Toolwin", "Shows the Tool Window", true,
        59, ref contextGUIDS,
        (int)vsCommandStatus.vsCommandStatusSupported +
        (int)vsCommandStatus.vsCommandStatusEnabled);
    Command command2 = commands.AddNamedCommand(addInInstance,
        "Load", "Load", "Loads a URL", true, 59,
        ref contextGUIDS,
        (int)vsCommandStatus.vsCommandStatusSupported +
        (int)vsCommandStatus.vsCommandStatusEnabled);
    // Add the menu item
    // This requires drilling down through the Microsoft
    // Office toolbar objects,
    // much of which is only partially documented :- (
    CommandBarPopup viewMenu = (CommandBarPopup)
        (applicationObject.CommandBars["MenuBar"]
        .Controls["&View"]);
    CommandBar viewMenuBar = viewMenu.CommandBar;
    CommandBarPopup othersMenu = (CommandBarPopup)
        (viewMenu.Controls["Oth&er Windows"]);
    CommandBar othersBar = othersMenu.CommandBar;
    command1.AddControl(othersBar, 1);

}
catch (System.Exception)
{
}
}
```

After this code comes the `OnDisconnection`, `OnAddInsUpdate`, `OnStartupComplete`, and `OnBeginShutdown` code. As with the sample in the previous section, I did not modify these methods and so I'm not listing them here, to save space. However, do not delete them after the wizard creates them, as you'll need them; otherwise your class will be abstract.

Next come the `QueryStatus` and `Exec` functions. Since I have two commands, I doubled them up in the `QueryStatus` function, checking if the `commandName` is either command. Here's the `QueryStatus` function:

```

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

```

The `Exec` function is remarkably simple. I check which command I received, and if its the `Show` command, I simply set the `toolwin` variable's `Visible` property to `True`. Or if the command I received is the `Load` command, I grab the command's parameter (passed in through the `varIn` object, which I cast to a string), and then I call the document object's `Navigate` command. Since I'm passing the command's parameter to the `Navigate` command, that means I'm expecting a URL for the parameter. Here's the code:

```

public void Exec(string commandName,
    EnvDTE.vsCommandExecOption executeOption,
    ref object varIn, ref object varOut, ref bool handled)
{
    handled = false;
    if(executeOption == EnvDTE.vsCommandExecOption.
        vsCommandExecOption.DoDefault)
    {
        if (commandName == "ToolWindowAddin.Connect.Show")
        {
            toolwin.Visible = true;
            handled = true;
            return;
        }
        if (commandName == "ToolWindowAddin.Connect.Load")
        {
            if (varIn != null)
            {
                string str = (string)varIn;
                toolwin.Visible = true;
                object emptyobj = null;
                doc.Navigate(str, ref emptyobj, ref emptyobj,
                    ref emptyobj, ref emptyobj);
                handled = true;
            }
        }
    }
}

```

```

        return;
    }
}
}
private _DTE applicationObject;
private AddIn addInInstance;
}
}

```

That's the whole add-in. To use it, compile it, start a new instance of Visual Studio .NET, and load the add-in. When it loads, you will see a tool window load, and it will start by going to the Web site www.wiley.com. (If you're not online, you'll get the standard Internet Explorer error message that the browser can't find the domain.)

Next, notice the two additional features I included. First, if you close the tool window, you can bring it back just like you can any other tool window, by visiting the View menu. In this case, choose View → Other Windows → ToolWin. (Presently, the icon is the one that add-ins use by default, a smiley face. In Chapter 8, in the section "Registering a Satellite DLL for Resources," I talk about other possibilities for icons.)

Now that you've been through the code, you can see how this menu item works: When you choose it, the IDE sends the command `ToolWindowAddin.Connect.Show` to the add-in. The add-in's `Exec` function kicks in, and displays the tool window.



The `Show` command has an interesting oddity: When you call any command in the add-in, the first thing the IDE does is display your tool window *for you*. So, technically, your `Exec` function doesn't have to do anything at all for the `Show` command, as long as the command is registered and the `QueryStatus` function says the command is available. But I thought leaving the `Show` command out of the `Exec` function would make for some strange code that would be confusing to someone who might be given the chore of maintaining my code for me, so I included the `Show` command in the `Exec` function.

As for the second feature, if you open up a command window by choosing View → Other Windows → Command window, you can type in the `Navigate` command:

```
ToolWindowAddin.Connect.Load http://www.cnn.com
```

The tool window will open (if it's not already open) and load the site www.cnn.com, provided you're on the Internet. (If you right-click on the tool window, you'll see it's the same browser window used in Internet Explorer; you get the usual popup menu.)

Though this code is handy for accessing the add-in, you might not want to have to open the command window every time. So here's a macro you can use that will prompt you for the Web address and then run the command for you:

```

Sub OpenSite()
    Dim url As String
    Dim customin, customout As Object
    url = InputBox("Enter the URL:")
    If (url = "") Then
        Exit Sub
    End If
    Dim cmd As Command =
DTE.Commands.Item("ToolWindowAddin.Connect.Load")
    DTE.Commands.Raise(cmd.Guid, cmd.ID, url, customout)
End Sub

```

Using the Form Designer with a Tool Window

In the previous section I showed you how to create a tool window that displays an ActiveX control, how the commands can manipulate the tool window, and how you can have a menu item that displays the tool window. In this section I'll show you one more important action: how to use the form designer to create your tool window, so you're not limited to displaying strictly ActiveX controls (and so you don't have to be an expert in ActiveX).

Although a tool window must display an ActiveX control, you can employ a little trick to get it to display a form that you build with the Visual Studio .NET IDE's Designer. Someone at Microsoft built what's called a *shim control*, an ActiveX control that serves one purpose: to display a form.



Actually, a shim control doesn't display an instance of the `Form` class; it displays an *instance* of a user control. However, the Designer lets you draw controls on a user control in the same way you would a form, so the end result is the same: You still get to use the Designer.

But before you can try this out, you need to obtain the shim control. Here's how: Visit <http://msdn.microsoft.com/vstudio/downloads/samples/automation.asp> and download the sample called ToolWindow Add-ins. (The file is a self-extracting executable.) Extract the files in it and open the solution in the VSUserControlHost directory. Build the project, then close the solution and return to your own solution. (If you want, instead of loading the solution, you can try out the command-line version of Visual Studio .NET, which will build a project for you without having to close the current solution. Go the Visual Studio .NET command prompt, change to the directory containing the VSUserControlHost.sln file, and issue the following command to do so.)

```
devenv VSUserControlHost.sln /build DEBUG
```

This library you are building is the ActiveX control that you'll be using to display your .NET form. When you build the library, the build process automatically registers the ActiveX control in the Registry so you won't have to.

Now start your new add-in project. I called my project ToolWinForms. Again, for this sample I used C#, and I turned on the wizard option regarding the creation of a “tools” menu so that I can have an `IDTCommandTarget` interface.

After you create the new project, make sure you add a reference to the library you just built. Right-click the project’s name in the Solution Explorer and choose Add Reference. In the Add Reference box, click the COM tab. Scroll down and double-click VSUserControlHost 1.0 Type Library, then click OK. This will add a reference so that you can access the ActiveX control that will be hosting your .NET form.

Again right-click the project’s name in the Solution Explorer; in the popup menu choose Add → Add User Control. In the Add New Item dialog box, make sure User Control is selected and type a name for your control, followed by .cs to denote the filename. (I called mine `ControlForm`, even though it’s technically not a form. The IDE will create the new class for you, and the Designer will open. Unlike a typical form, the Designer will show only a grid where you can drop controls, no form features. Open the toolbox by choosing View → Toolbox.

Next, follow these steps for dropping the controls on the form:

1. Widen and lengthen the form so its size is about 600 pixels wide by about 200 pixels tall. (You can see its size by right-clicking it and choosing Properties; in the Properties window scroll down to the Size property.)
2. Double-click the Panel in the toolbox. A panel will appear in the form. Right-click the panel and choose Properties; the Properties window will open. Scroll down until you see Dock; set the Dock property to Fill. (To do so, either type an F, or click the drop-down arrow and click the box in the center of the small window that opens.)
3. Double-click the button in the toolbox. Slide the button toward the upper-left corner. Set its Text property to Command.
4. Double-click ListView in the toolbox. Drag it to the left edge, just below the button. Drag its bottom toward the lower edge of the form. Drag its right edge to the right edge of the form.
5. Now you can add the columns for ListView. Right-click on ListView on your form and choose Properties. Set the View property to Details. Also in the Properties window, scroll down and click the Columns property. A small button with an ellipsis (...) on it will appear; click it. The ColumnHeader Collection Editor will open. Follow these substeps:
 - a. Click Add to add a new column. Set the Name property to NameColumn, the Text property to Name, and the Width property to 150.
 - b. Click Add to add a second column. Set the Name property to IDColumn, the Text property to ID, and the Width property to 80.
 - c. Click Add to add a third column. Set the Name property to GUIDColumn, the Text property to GUID, and the Width property to 250.
 - d. Click OK.

Your form is now set up, and it should look more-or-less like the one shown in Figure 7.2.

Name	ID	GUID

Figure 7.2 The finished form.

Double-click the button so you can add an `OnClick` handler. Here's the code for the handler:

```
private void button1_Click(object sender, System.EventArgs e)
{
    foreach (Command cmd in Connect.applicationObject.Commands)
    {
        ListViewItem lvitem = new ListViewItem(cmd.Name, 0);
        lvitem.SubItems.AddRange(new String[] {cmd.ID.ToString(),
            cmd.Guid.ToString()});
        listView1.Items.Add(lvitem);
    }
}
```

You will also need to add a *using* line for the `EnvDTE` toward the top of the code. Here's the top of my file so you can see where I put it:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;

namespace ToolWinForms
{
    // Added this so I can access the IDE.
    using EnvDTE;
```

The rest of the code I left unchanged, so I'm not including it here.

On to the `Connect.cs` file, which was generated automatically by the wizard. Once again I needed a new GUID for the tool window, which I'll use in the `OnConnection` function to register my new tool window. To create the GUID, I again used `Tools → Create GUID`.

And as with the example in the preceding section, I'll need a reference to the `System.Windows.Forms` assembly. However, by using the Designer, the IDE added this reference for me automatically.

Here's the start of my `Connect.cs` code. You can see I again added a line for the `System.Windows.Forms` assembly:

```
namespace ToolWinForms
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;
    using System.Runtime.InteropServices;
    using EnvDTE;

    // Added this...
    using System.Windows.Forms;
```

Next is the header for the class. Like the example in the preceding section, I'm allocating a variable for my object. However, instead of using a Web browser object, this time I'm using the object defined in the `VSUserControlHost` library that I built at the start of this section. You can see the type of the object is `IVSUserControlHostCtl`.

```
[GuidAttribute("CBC8C226-9EDE-405A-8D3B-3ACDAEDBFF2C"),
ProgId("ToolWinForms.Connect")]
public class Connect : Object,
    Extensibility.IDTExtensibility2, IDTCommandTarget
{
    private VSUserControlHostLib.IVSUserControlHostCtl doc = null;
    private Window toolwin = null;

    public Connect()
    {
    }
}
```

Here's the `OnConnection` function, which is similar to the `OnConnection` function in the preceding section; however, in the `CreateToolWindow` call I passed the name of the ActiveX control for this example, which is `"VSUserControlHost.VSUserControlHostCtl"`. As before, I saved the return document, but this time cast it to the `IVSUserControlHostCtl` type, saving it in the `doc` object.

Next, I tell the `doc` object the name of my form class. However, to do that, I must locate the assembly containing the form class (which is the currently running assembly). This involves a call to `GetExecutingAssembly`. Then I take the results and pass them along with the name of my form class to the `doc` object by calling the `HostUserControl` member function.

This time I create only a single command and I set its description to "ToolWinForms", which will ultimately appear on the View⇨Other Windows menu. I then use the same convoluted process as in the example in the preceding section to set up the menu:

```

public void OnConnection(object application,
    Extensibility.ext_ConnectMode connectMode,
    object addInInst, ref System.Array custom)
{
    applicationObject = (_DTE)application;
    addInInstance = (AddIn)addInInst;

    try
    {
        String newguid = "{743F0A16-B976-40f2-9578-87BB53C2DCDA}";
        object tempdoc = null;
        Window macroexp = applicationObject.Windows.Item(
            Constants.vsWindowKindMacroExplorer);
        String objkind = macroexp.ObjectKind;

        toolwin = applicationObject.Windows.CreateToolWindow(
            addInInstance,
            "VSUserControlHost.VSUserControlHostCtl",
            "Form Host", newguid, ref tempdoc);
        toolwin.Visible = true;
        doc =
(VSUserControlHostLib.IVSUserControlHostCtl)tempdoc;
        System.Reflection.Assembly asm = System.Reflection.
            Assembly.GetExecutingAssembly();
        // Be sure to set the right namespace.form name
        // in this next line.
        doc.HostUserControl(asm.Location,
            "ToolWinForms.ControlForm");
    }
    catch (System.Exception e)
    {
        MessageBox.Show(e.Message);
    }

    object []contextGUIDS = new object[] { };
    Commands commands = applicationObject.Commands;

    try
    {
        // If the commands already exist, this whole block will
        // drop out to the catch block below, so I don't need
        // to worry about accidentally adding anything twice.
        Command command1 =
commands.AddNamedCommand(addInInstance,
            "Show", "ToolWinForms", "Shows the Tool Window", true,
            59, ref contextGUIDS,
            (int)vsCommandStatus.vsCommandStatusSupported +

```



```

        (int)vsCommandStatus.vsCommandStatusEnabled);
    CommandBarPopup viewMenu = (CommandBarPopup)
        (applicationObject.CommandBars["MenuBar"]
        .Controls["&View"]);
    CommandBar viewMenuBar = viewMenu.CommandBar;
    CommandBarPopup othersMenu = (CommandBarPopup)
        (viewMenu.Controls["Oth&er Windows"]);
    CommandBar othersBar = othersMenu.CommandBar;
    command1.AddControl(othersBar, 1);
    }
    catch (System.Exception)
    {
    }
}

```

The `QueryStatus` and `Exec` functions are not as interesting as the example in the preceding section. For the `QueryStatus` function, I just check for the single `Show` command and issue a status that it's supported and available. For the `Exec` function, I again check for the single `Show` command and then make the tool window visible. However, following the `QueryStatus` and `Exec` functions, I did make a small change to the two variables that the wizard automatically added; I made them static. The reason is that I want to be able to access them from the form class and I don't want to worry about having an instance of `Connect` handy. Since I'll only be creating one instance of `Connect`, I went ahead and made the two variables static.

```

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

public void Exec(string commandName,
    EnvDTE.vsCommandExecOption executeOption,
    ref object varIn, ref object varOut, ref bool handled)
{
    handled = false;
    if(executeOption == EnvDTE.vsCommandExecOption.
        vsCommandExecOptionDoDefault)

```

```

        {
            if (commandName == "ToolWinForms.Connect.Show")
            {
                toolwin.Visible = true;
                handled = true;
                return;
            }
        }
    }

    // I made these static so the form can access them
    // without an instance of Connect.
    static public _DTE applicationObject;
    static public AddIn addInInstance;
}
}
}

```

That's it. Now the source files are all ready, and you can build the program. After you build it and start a new instance of Visual Studio .NET, you can start the add-in using the Add-in Manager. When the add-in begins, it will install a menu item as View→Other Windows→ToolWinForm. From then on, you can access the add-in through this menu item, rather than through the Add-in Manager.

When you start the add-in, its tool window will open, showing the layout that you built with the Designer. Initially, the tool window might be the smaller, but you can resize it to make it bigger; or you can drag it onto one of the other tool windows to dock it.

When you click the button labeled Command, the ListView control will fill with all the commands the IDE knows. (If you tried any of the macros that list commands, you can see that this compiled add-in is much faster than a macro.)

Moving Forward

In this chapter I took the concepts presented in Chapter 6, "Introducing Add-ins," and built actual add-ins to demonstrate them. In the process I showed you the different ways you can feature a user interface with your add-in, including using tool windows. I'm partial to tool windows, because by putting your GUI in a tool window, the add-in takes on the familiar look and feel of the Visual Studio .NET IDE, allowing your IDE users to easily use the add-in without having to learn a new GUI. Plus, the tool window add-in makes the add-in seem fully integrated to Visual Studio .NET.

In the next chapter I continue the discussion of add-ins, covering three distinct topics related to them: the life cycle of an add-in, how to debug an add-in, and how to create satellite DLLs to allow for globalization of your software.

Life Cycles, Debugging, and Satellite DLLs

In this chapter I discuss three important topics: the life cycle of an add-in, how to debug an add-in (including a command-line add-in), and how to create satellite DLLs for your add-in that allow your add-in to display information in a user's native language.

The Life Cycle of an Add-in

The `OnConnection` handler of an add-in receives a `connectMode` parameter that describes the current state of the add-in. For example, if the `connectMode` parameter is `ext_cm_UISetup`, then your add-in code can assume this is the first time the add-in has been run since the add-in was installed. In that case, you can do initial processing such as adding commands, menu items, and toolbar buttons.



The values for the `connectMode` variable, such as `ext_cm_UISetup` and `ext_cm_Startup` are found in the `Extensibility.ext_ConnectMode` enumeration.

Here are the possible values for the `connectMode` parameter:

`ext_cm_AfterStartup` (equal to 0). Your add-in will receive this when the IDE user loads the add-in manually through the Add-in Manager.

`ext_cm_Startup` (equal to 1). If your add-in is set to load when the IDE starts, the add-in will receive this value.

- ext_cm_External (equal to 2).** If an external program or component forced the add-in to load, the add-in will receive this value. This would happen, for instance, if an automation controller launches a new instance of Visual Studio .NET using a `CreateObject` call.
- ext_cm_CommandLine (equal to 3).** At the time of the writing, this option was not available. When you load the command-line version of the devenv program, your add-in will receive the `ext_cm_Startup` value.
- ext_cm_Solution (equal to 4).** If you have a solution that requires a particular add-in and you load that solution, causing your add-in to load, your add-in will receive this value.
- ext_cm_UISetup (equal to 5).** The first time you run the add-in after it installs, your add-in will receive this value.

When you create an add-in using the Add-in wizard, and you check the box for adding a tool button, you automatically get code in your `OnConnection` handler that responds to the `ext_cm_UIStartup` value. If you need to force an `ext_cm_UISetup` value, you have two ways available:

- You can run the command-line command `devenv /setup`.
- You can set the `CommandPreload` registry entry.

To set the `CommandPreload` Registry entry, open the Registry editor, and move down to `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.0\AddIns` and find your add-in. Under your add-in, you will see a `CommandPreload` entry. Change its value to 1.



I prefer not to use `ext_cm_UISetup`. The reason is that any changes I make to the UI, such as adding a menu item, are then tied strictly to `ext_cm_UISetup`. However, if the IDE user is running multiple instances of Visual Studio .NET (as when I'm writing an add-in) the add-in's menu setup can get overwritten by another instance of Visual Studio .NET, causing the menu addition to go away. To prevent that, I make a simple assumption: If the commands my add-in adds are not present, the menus it adds are probably not present either. So I put all my setup inside a Try/Catch block; inside the try block, I *first* try adding my new commands before adding the menu items. The IDE has a feature whereby it will throw an exception if a command already exists. I then do nothing in my catch block, allowing the add-in to continue. But if the exception does not get thrown, I add the menu items. This process occurs every time the add-in loads.

Debugging an Add-in

Debugging an add-in involves starting a second instance of Visual Studio .NET, in which you run your add-in and debug it. You can set breakpoints in your add-in (such

as in the `OnConnection` method), and when you interact with the add-in in the second instance of Visual Studio .NET, the second instance will halt when it encounters your breakpoint. You can then debug from the first instance of Visual Studio .NET.

To debug the add-in, make sure the project is set up to run Visual Studio .NET when you begin a debug session. I describe these necessary steps in “Building and Running the Add-in” in Chapter 6, “Introducing Add-ins.”



The Visual Studio .NET IDE has two methods of debugging: one that debugs strictly VB.NET and C# programs and one that debugs C++ programs. If you develop your add-in in C++, and you run the debugger, the debugger will take a lot longer to load than it would if you were debugging in C# and VB.NET.

When you’re ready to debug your add-in, you can start it in the debugger by making sure it’s the startup project (right-click its project in the Solution Explorer; and in the popup menu, choose Set as Startup Project) and then simply pressing F5. This will launch the second instance of Visual Studio .NET. You can then set breakpoints as you normally would. But to debug the add-in itself, you need to load the add-in. To load the add-in, open the Add-in Manager and check the box to the left of your add-in’s name. When you click OK, the add-in will load.

Keeping the Registry Clean

If your add-in makes any modifications to the Registry, some of these modifications will not stick. Visual Studio .NET updates the Registry when it exits; thus, when the second instance (the one being debugged) changes the Registry and exits, you will return to the first instance. When that instance exits, it will again update the Registry, undoing some of the changes from the second instance. As for which changes it undoes, specifically they include those dealing with the creation of commands, the creation of toolbar buttons, and the creation of menu items. Therefore, be aware that your commands may come and go as you’re debugging.

Moreover, the changes to the Registry that *do* stick can be problematic. If, say, while in the second instance of Visual Studio .NET you change whether your add-in loads when Visual Studio .NET starts, this change will stick, so when you finish debugging and return to the first instance of Visual Studio .NET, the change will be there. If you’re not aware of this, and you shut down Visual Studio .NET for the day, when you start it back up the next morning, your add-in will load. If the add-in is in a volatile state, the Add-in manager might, to your surprise, detect an exception and issue a warning, ask whether you want to unregister the add-in. This is not a problem, exactly (although I recommend clicking No), but it can be a source of confusion, especially if you are working with other developers who might not be as familiar with add-in development as you are.

Debugging the Command-Line Version

You can start up the Visual Studio .NET product in command-line mode, allowing you to do batch builds. In this section I show you how to set up the IDE so it can spawn your add-in in command-line mode, allowing you to debug the add-in in that mode as well.



Be careful not to confuse the two terms *command-line mode* and *commands*. When you operate the Visual Studio .NET tool in command-line mode, you type its executable name, `devenv`, at the DOS command prompt, along with several parameters. This prevents the IDE from opening, allowing `devenv` to write results to the DOS window. *Commands*, on the other hand, refer to the commands available to you from within the IDE. When you click a toolbar button or choose a menu item or type a name into the Command window within the IDE, you cause the IDE to execute one of its many commands.

When you build an add-in, you need to take into consideration whether the add-in can be used during a command-line build. Although you personally might not use the command-line version of Visual Studio .NET, it's very possible the users of your add-in will, especially if you're developing a commercial add-in.



Although this may seem incorrect, in the first version of Visual Studio .NET (considered Visual Studio version 7.0, which, at the time of this writing is the current version), in order for your add-in to load when the command-line version of `devenv` loads, you must set the add-in to load when the IDE starts, by choosing the Startup option in the Add-in Manager for your add-in. If you simply select Command Line, the add-in will not load when the command-line version of `devenv` starts. In fact, whether you check Command Line or not, the add-in will still load if you have the Startup option selected.

Debugging the Command-Line Add-in

To instruct the IDE to launch `devenv` in command-line mode, you need to set up the debugger to add the appropriate command-line options to the `devenv` command. That will force `devenv` to run in command-line mode.

To set up these items, do the following:

1. Open the Properties window for the add-in project.
2. In the Debugging section under Configuration Properties, make sure the external program is set to `devenv`. (It should be.)
3. For the command-line arguments, type the name of a solution you wish to build in command-line mode. (Remember, command-line mode is primarily for building solutions. I do not, however, recommend that you make this solution the same as the one containing your add-in.) For mine, because I was building the `VSUserControlHost.sln`, I set up the following command-line argument:

```
VSUserControlHost.sln /build DEBUG
```

4. Set the working directory to the location of the solution file you want to build.



When you debug in command-line mode, you are somewhat restricted, in that you won't see the DOS command window. If you're adventurous and you want to try to see the output, you can play around with the DOS cmd command and its /K and /C parameters. For my work, however, I primarily rely on setting breakpoints inside my add-in and debugging from there.

Setting Up Multiple Debug Configurations

If you write an add-in that you expect to be used with both command-line mode and IDE mode, you can make your debugging sessions simpler by creating additional configurations for your project. By default, your projects have two configurations: Debug and Release. (This is true whether you're using C++, C#, or VB.NET.)

If you followed the instructions I gave earlier in this section, you now have your add-in set up such that when you run it, the second instance of devenv will launch in command-line mode. But what if you want to launch it in IDE mode? It would be annoying to have to open up the project settings each time you want to launch in a different mode. Instead, I recommend you create a separate configuration. The following steps describe how to create and set up this new configuration:

1. Open up the Properties dialog box for your add-in project. Depending on which language you're using, your Property Pages dialog box will look slightly different. But regardless of the language, your Property Pages dialog box will have a drop-down listbox at the top where you can choose a configuration, as well as a button called Configuration Manager.
2. Click the Configuration Manager button. The Configuration Manager will open, as shown in Figure 8.1.

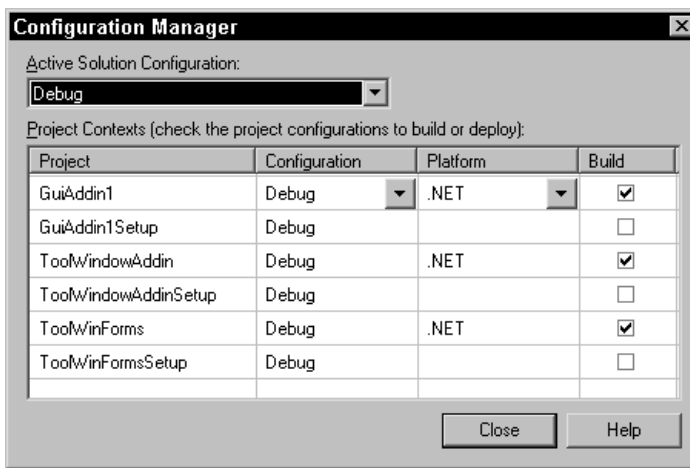


Figure 8.1 The Configuration Manager.

3. In the Configuration Manager, click the drop-down listbox at the top, labeled Active Solution Configuration, and choose New. A small dialog box called New Solution Configuration will open, as shown in Figure 8.2.
4. In the edit control labeled Solution Configuration Name, type the name CmdDebug (or any name to notate the debug version of the command-line mode; I prefer to keep the name short, thus my choice of an abbreviated name, CmdDebug).
5. Next, click the drop-down listbox labeled Copy Settings From and choose Debug, as you want to create the new configuration initially as a duplicate of the Debug configuration. (In the next step, you'll modify the configurations so two won't be identical.) Make sure the "Also create new project configuration(s)" checkbox has a check beside it, and click OK.
6. When the New Solution Configuration dialog box closes and you're back in the Configuration Manager, click Close.



The changes you make by using the Configuration Manager are for the entire solution, not just a single project.

After the Configuration Manager closes, you'll be back in the Property Pages dialog box for the project. The Configuration drop-down listbox will now have the new CmdDebug configuration, which is the same as the Debug configuration. Since for most projects you'll spend most of your time working in IDE mode rather than command-line mode, I now recommend changing the Debug configuration to run in IDE mode, leaving the CmdDebug configuration to run in command-line mode. To make the Debug configuration run in IDE mode, choose Debug in the Configuration drop-down listbox and delete any text inside the edit control labeled "Command-line arguments." Also, go ahead and clear out the Working Directory textbox. (By removing these command-line arguments, devenv will launch in IDE mode.)



If you plan to execute devenv without running the debugger, you can repeat all these steps for Release configuration, creating a CmdRelease configuration that contains the command-line options that result in the command-line execution, while leaving the command-line options blank for the Release configuration.

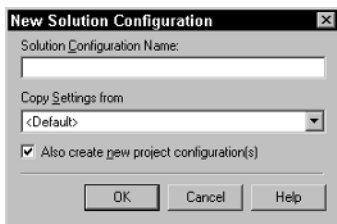


Figure 8.2 The New Solution Configuration.

ADD-INS AS COM COMPONENTS AND ASSEMBLIES

When you create an add-in, you are actually creating a COM component that also serves as an assembly. Here's how this works: You create two files, an assembly, in the form a DLL, and a type library with a .tlb extension. (The Add-in wizard sets up your project to create both of these files for you when you build your project.) The Visual Studio .NET IDE expects not an assembly, but a COM file for the add-ins. The add-ins you are building with Visual Studio .NET are actually assemblies, with a COM wrapper around them (in the form of a type library), which allows your assemblies to serve as COM components, satisfying the COM requirement. With COM components, you use one approach for building satellite DLLs, as I describe in the following sections; with assemblies, you use a different approach for building satellite DLLs, which I also describe.

So which approach should you use? The online help suggests you use the COM approach. However, doing so requires stepping out of the .NET world into the COM world, which I prefer not to do. (I am, after all, writing my add-ins for .NET.) Therefore, in this chapter I show you how you can ignore the online help instructions and use the .NET's assembly form of creating satellite DLLs.

You can then close the Property Pages dialog box by clicking OK. Now you can choose whether you want to run `devenv` in command-line mode or in IDE mode. If you look up at the standard toolbar (if the standard toolbar isn't present, right-click on the toolbar area and choose Standard), you'll see a drop-down listbox with the configuration names Debug, CmdDebug, Release, and, possibly, CmdRelease, depending on whether you added the CmdRelease configuration. To choose a configuration, simply make your selection from this configuration drop-down listbox.

After you choose your configuration, when you run the debugger, you'll either get the command-line version of `devenv` or the IDE version of `devenv`, depending on your selection in the drop-down listbox.



If you're writing your add-in in C++, be careful when playing around with different configurations, because each time you build under a different configuration, Visual Studio .NET registers the resulting DLL as the server for your add-in's COM component. Since each configuration gets its own directory in your project directory, and this directory contains a resulting DLL, the location of your COM component will effectively change each time you change to a different configuration. Fortunately, the IDE runs a registration program, to register the DLL you are building; this registration will keep the COM system updated on where your latest DLL is.

Creating a Satellite DLL for Resources

When you want to include resources in your add-in's DLL (such as graphics, icons, and strings) that are culture-specific, you can put the resources in a satellite DLL. But before

I explain what this is all about, I direct your attention to the sidebar titled “Add-ins as COM Components and Assemblies.”



If you’re already familiar with building .NET assemblies and satellite DLLs, this note is for you. You might have noticed a seeming discrepancy in the online help for add-ins (which I also mention in the sidebar): It reads that satellite assemblies use the *ID codes* such as 1033; but the online help for assemblies refers to *culture names*, such as en-us. The reason is that add-ins are COM components, which use the IDs, with assembly wrappers around them. After I explain about cultures in this section, I show how you can use the assembly form of the satellites to make all your work .NET-compliant.

About Culture-Specific Information

When Windows NT version 3.1 first shipped in the early 1990s, it contained an API called the National Language Support, which provided programmers with a way to determine a user’s culture-specific information, such as the name of the language, the currency format, time and date formats, and the names of the days and months. By accessing culture-specific information in your programs, you can allow users to interface with familiar words and formats, rather than forcing them to work either in English, or any other language for that matter, which might not be their native language.

Now with .NET, Microsoft includes National Language Support through the `System.Globalization` namespace. This namespace includes a class called `CultureInfo`, which holds information specific to a culture, including a string identifier, which names the culture, and a numeric identifier called LCID, which stands for locale identifier, or if you prefer, local culture identifier.



If you care about standards, the string identifiers are specified in the ISO standards 639-1 and 3166; the format for the string is specified in the standard RFC 1766.

Examples of the string identifier are:

en for English

en-US for American English

en-AU for Australian English

en-JM for Jamaican English

es-MX for Mexican Spanish

And for every language, the Globalization namespace includes both a generic form of the language, such as en for English and fr for French, as well as national versions of the languages, such as en-CA for Canadian English and fr-CA for Canadian French. The reason for this distinction is that, for example, Canadian French is slightly different from

the French spoken in France, and Canadian English is slightly different from the English spoken in the United Kingdom.

The numeric identifiers were specified in the original National Language Support API that Microsoft used in Windows NT. For each string identifying a culture, the API has a single numeric identifier. The identifier for American English, for example, is 1033. You may have seen these numbers before if you installed the version of the Microsoft Developer Network (MSDN) that accompanies Visual Studio .NET, where you may have noticed a directory that is simply a number. On my system, that number is 1033, since I installed the American English version of the software. Here's where mine is installed:

```
C:\Program Files\Microsoft Visual Studio .NET\Msdn\1033
```

This directory contains the American English version of the MSDN online help files.

The 1033 identifier also appears as a directory name in numerous other places in my installation. If I open a command (DOS) prompt, and move to the directory C:\Program Files\Microsoft Visual Studio .NET, and type `dir 1033 /s`, I see 294 directories named 1033. Each of these contains culture-specific information. But with the assemblies you build in .NET, the situation is slightly different. Instead of using a number such as 1033, .NET uses the string version of the culture name, such as en-US.

Inside each of these culture directories are those files for which Microsoft decided to provide culture-specific versions. As for obtaining the month and currency formats, which I prefer in the American version of English, the various software packages installed in C:\Program Files\Microsoft Visual Studio .NET can simply access the System.Globalization namespace. But, often, as a programmer, you don't even need to directly use the classes in the System.Globalization namespace; instead, you can simply write out a string—for example, as currency—and the .NET framework will automatically format the string properly for the local culture.

How does the .NET framework know the local culture? When the user installs Windows, he or she specifies the preferred culture. In the Control Panel is an icon named Regional Options. When you double-click this entry, you can choose your preferred culture. Additionally, you can customize the entry by choosing a different setting for numbers (for example, if you prefer to use commas or decimals as thousand separators), currency (if you prefer to use, for example US\$ instead of \$ for United States currency), time format (such as 8:45:56 A.M. or 08:45:56), and the date format (such as 10/27/02 or 27/10/02 for the twenty-seventh day in October 2002).

Valid Culture Identifiers

Here's a macro that will list all the culture identifiers known to .NET. First, include the following line at the top of the macro module:

```
Imports System.Globalization
```

Now here's the code for the macro. It lists the culture ID, the string ID, and the full name of each culture.

```

Sub ListCultures()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim ci As System.Globalization.CultureInfo
    For Each ci In CultureInfo.GetCultures(CultureTypes.AllCultures)
        VBMacroUtilities.Print( _
            ci.LCID.ToString().PadRight(8) & _
            ci.Name.PadRight(12) & ci.EnglishName)
    Next
End Sub

```

Here are the some of the output lines:

10	es	Spanish
2058	es-MX	Spanish (Mexico)
3082	es-ES	Spanish (Spain)
4106	es-GT	Spanish (Guatemala)
5130	es-CR	Spanish (Costa Rica)
6154	es-PA	Spanish (Panama)

You can use this program to determine the string ID of cultures. (You will need this information in many of the following sections.)



You can find out the current culture by checking the `CurrentUICulture` property of the `CultureInfo` class. The property is an instance of `CultureInfo`. It is also a static member; therefore, you do not need to start with an instance of `CultureInfo`.

Adding Culture-Specific Resources in .NET

Since adding culture-specific information to a program in .NET is not a topic covered in many books, in this section I show you how to create multilingual resources in a .NET program that is *not* an add-in. Then, in the next section, I show you how to apply these techniques to your add-ins.

Forms and Multiple Languages

The Visual Studio .NET Form Designer has a little-known feature that makes adding languages to your forms incredibly easy. To try out this feature, open the New Project dialog box and click Visual Basic Projects, then choose Windows Application. Create the new project and open up the form, if it's not already open.

Now try out the following steps:

1. Right-click the form, and in the popup menu choose Properties to open the Properties window for the form. (For the remaining steps, I'm assuming your Properties window is listed by category. To make sure it is, click the upper-left icon in the Properties window, the one called Categorized.)

2. In the Properties window, scroll down to the Appearance section and set the Text property to Hello!.
3. Scroll down to the Misc section. Inside this section, you will see two important properties: Language and Localizable. Set Localizable to True; this will make your form multilingual.
4. Also in the Misc section, click the Language property and you'll see a drop-down list. In the drop-down list, choose German. Now scroll back up to the Text property and change it to Guten Tag!.

Now the form gets interesting: Keep an eye on the caption of the form, and in the properties window scroll back down to the Language property. Change it back to Default. Now look again at the caption of the form: It changed back to Hello!. You now have a multilingual form. You can also add controls to the form (I recommend adding them while the language is set to Default; otherwise you'll get a warning message reminding you to set the default language). Then when you switch between languages, you can also change the text on the controls as well. Note that you can only change the language from the form's property settings, and when you do, the language changes for all the controls on the form.

Go ahead and save the form you were working on. Next, build the project. You can then run the program if you want, but what I mainly want you to see is the set of resulting files and assemblies. Open up either a Windows Explorer window or a DOS prompt and go to the directory containing the project you've been working on; in that directory, move to the bin subdirectory. Inside the bin subdirectory, you can see the executable file for the program you built. Beside this file is a directory called de (which stands for Deutsch, German for "German"). Drop down into the de directory, where you'll see a DLL. This DLL's filename will be the same as your project name, followed by .resources.dll. This DLL contains the regional information for the German language. In other words, this DLL is a cultural assembly.

Now move up two directories to the main project directory. If you look at the files here, you will see two resource files with the extension .resx (which stands for Resource/XML):

- Form1.de.resx
- Form1.resx

The first file contains the German version of the form; the second file contains the default version of the form, in which I placed the English version of the form, as is customary (at least for Americans).

Adding Cultural Assemblies in General

In the previous section you used the form designer to create a German language assembly. In this section I show you how to do the same thing but without using the form designer.

It turns out the IDE is culture-smart. If you set up your resources properly (as the form designer did for you automatically in the previous section), when you build your

project, the IDE will automatically transform your culture-specific resources into its own satellite DLLs. To see this in action, perform the following steps:

1. Create a new VB.NET or C# project, but this time use the Console Application template.
2. After the IDE creates the project for you, right-click the project name in the Solution Explorer; in the popup menu choose Add→Add New Item. The Add New Item dialog box will open.
3. In the Add New Item dialog box, choose the Local Project Items category, then choose the Assembly Resource File template. (*Don't* click Open yet!)
4. This step is important: For the name of the resource, type: `words.de.resx`.
5. *Now* click Open. The Add New Item dialog box will close and the resource editor will open inside the main IDE.

Before proceeding, I want to explain why the name is important: In order for the IDE to recognize that your resource file is culture-specific, the filename must consist of some name, followed by a period, followed by a legitimate culture string, followed by a period, followed by the extension `resx`, like so:

```
name.<Culture string identifier>.resx
```

Examples are `name.de.resx` for German, and `name.es-MX.resx` for Mexican Spanish.

When you follow the naming convention properly, Visual Studio .NET will automatically compile your culture-specific resources into its own satellite DLLs when you build your project. It will also put the satellite DLLs inside subdirectories named for the language. The `name.de.resx` resource file will go in a directory called `de`, and the `name.es-MX.resx` resource file will go in a directory called `es-MX`.



In order for a culture-specific resource to end up in its own satellite assemblies, the name must include a valid culture string identifier. If you make one up, such as `name.something.resx`, the IDE will not compile the resource separately. Instead, the IDE will compile the resource into the main project assembly.

But before you build, continue working on the project I began describing in the preceding steps:

6. Open up the `words.de.resx` file if it's not already open. In the resource editor, click the first line to add a column, as shown in Figure 8.3.
7. In the name column, type the word `GREETING`. In the value column, type the words `Guten Tag!` This is shown in Figure 8.4. Now press Enter to add the new row. Choose File→Save `words.de.resx`.
8. Repeat the preceding steps to add a resource file called `words.es-MX.resx`, which represents Mexican Spanish. Again type `GREETING` for the name column. (This is the keyword by which you will locate the language-specific string.) In the value column, type the word `Hola`.

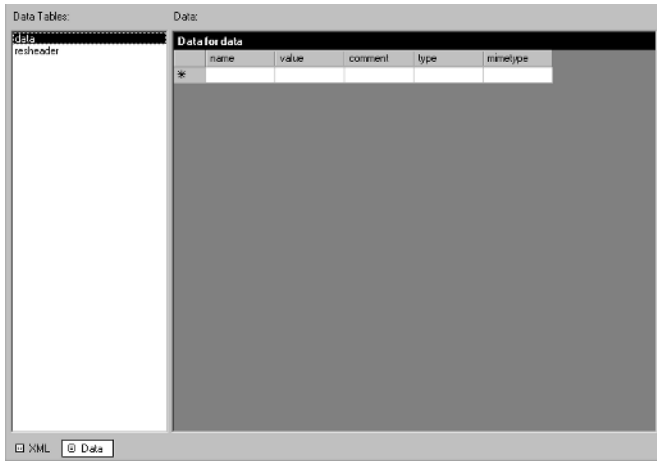


Figure 8.3 Click the first line in the resource editor to add a resource string.

9. Next you can add a default resource. Doing so is good policy, because it ensures that people whose language is one that you didn't include can at least read the English. To do this, create one more resource, but this time simply call it `words.resx`. Open it and add a single line, with `GREETING` for the name column and `Hello!` for the value column. Then save the file.

That completes the resources. Next you can access the proper string based on the current culture. Add the following code to your main module. (And, yes, there are some strange lines in it, which I'll explain after the code. But this really is the easiest way to access the resources.)

```
Imports System.Globalization
Imports System.Resources

Module Module1

    Private Class words
    End Class

    Sub Main()
        Dim rm As ResourceManager
        rm = New ResourceManager(GetType(words))
        Dim s As String = rm.GetString("GREETING")
        Console.WriteLine(s)
    End Sub

End Module
```

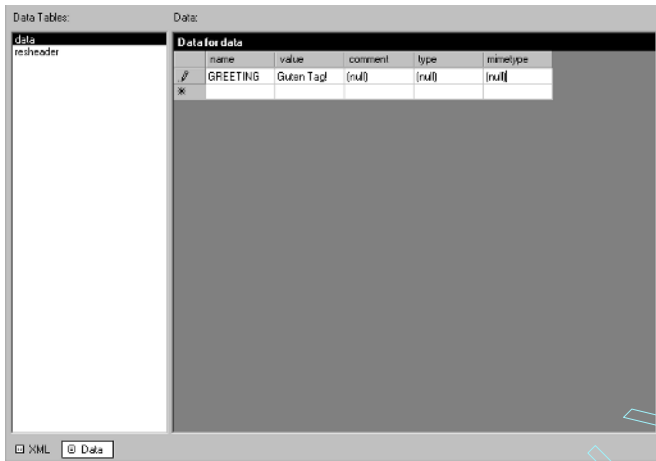



Figure 8.4 Add the words GREETING and “Guten Tag!” to the resource.

First, the empty class called `words` might seem pretty useless. But it serves one important purpose: The constructor for the `ResourceManager` object, which is the object you use for accessing the resources, needs a type whose name happens to match the first word in the filename of the resource files, before the culture name and `.resx` extension. Thus, in this case, with resource files name `words.de.resx` and so on, you need to create a type called `words`. The type doesn't have to actually do anything; you simply need a type with the proper name. (This means, of course, that you will want to use care in naming your resource files so that you can create a type name that is available. To that end, many people create a class and make that the main class in which they put all the processing. Then they name their resources the same as the class.)

You can see in the code that to create the `ResourceManager` instance, you pass the type instance of the `words` class using the `GetType` keyword. Then, after obtaining the `ResourceManager` instance, you can call `GetString` to retrieve the string resource. And this program, like the one in the previous section, is multilingual. If you change to a different cultural setting, you can read a different resource.

If you prefer to program in C#, here's the same program in that language. Notice that the C# equivalent of the VB.NET `GetType` keyword is `typeof`.

```
using System;

namespace CSMultilingualConsole
{
    class words
    {
    }

    /// <summary>
    /// Summary description for Class1.

```

```

/// </summary>
class Class1
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        System.Resources.ResourceManager rm = new
            System.Resources.ResourceManager(typeof(words));
        Console.WriteLine(rm.GetString("GREETING"));
    }
}

```

Making Your Add-ins Multilingual

In Chapter 6, in the section “Managing Add-ins,” I made brief mention of satellite DLLs in my discussion of the Registry. Under the Registry entry for each add-in, you can specify both a `SatelliteDLLName` entry and a `SatelliteDLLPath` entry. When Visual Studio .NET loads your add-in, it constructs the satellite name and path by taking the name stored in `SatelliteDLLPath`, appending a backslash, followed by the numeric identifier for the culture, appending another backslash, and finally appending the name stored in `SatelliteDLLName`. Thus, if the `SatelliteDLLName` is `MyResources.dll`, the `SatelliteDLLPath` is `c:\dev\MyAddIn` and your specific culture identifier is `2055` (which is the German language for Switzerland); then .NET will load the DLL given by `c:\dev\MyAddIn\2055\MyResources.dll`.

However, this information applies primarily if you’re writing a COM component that is not an assembly. But this book is about .NET, so I prefer to make use of the assembly features of satellite DLLs. Thus, for the add-in I’m about to show you, I’m going to use the procedures I described in the previous two sections.

Go ahead and create a new add-in. Make sure to select the option for the Tool button so that the add-in’s Connect class will be derived from `IDTCommandTarget`.

Following is the add-in’s main Connect code. In this code I add a command for the Show menu item, as I did in the Chapter 7, “Creating Add-ins for the IDE.” I also set up the add-in to use the `VSUserControlHost`, also described in Chapter 7. (Make sure you remember to add a reference to the `VSUserControlHost 1.0 Type Library`.)

```

namespace LocalizedAddin
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;

```

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

[GuidAttribute("5052EF2A-17F7-4C78-9FF4-668C3342060E"),
ProgId("LocalizedAddin.Connect")]
public class Connect : Object, Extensibility.IDTExtensibility2,
    IDTCommandTarget
{
    public Connect()
    {
    }

    public void OnConnection(object application,
        Extensibility.ext_ConnectMode connectMode,
        object addInInst, ref System.Array custom)
    {
        applicationObject = (_DTE)application;
        addInInstance = (AddIn)addInInst;
        object []contextGUIDS = new object[] { };
        Commands commands = applicationObject.Commands;
        _CommandBars commandBars = applicationObject.CommandBars;
        object objTemp = null;
        try
        {
            Command command =
commands.AddNamedCommand(addInInstance,
                "Show", "Time Form", "Shows the tool window",
                true, 59, ref contextGUIDS,
                (int)vsCommandStatus.vsCommandStatusSupported +
                (int)vsCommandStatus.vsCommandStatusEnabled);
            CommandBarPopup viewMenu = (CommandBarPopup)
                (applicationObject.CommandBars["MenuBar"]
                .Controls["&View"]);
            CommandBar viewMenuBar = viewMenu.CommandBar;
            CommandBarPopup othersMenu = (CommandBarPopup)
                (viewMenu.Controls["Oth&er Windows"]);
            CommandBar othersBar = othersMenu.CommandBar;
            command.AddControl(othersBar, 1);
        }
        catch(System.Exception /*e*/)
        {
        }
        String guidstr = "{10906CA9-2FB0-42ca-9BC2-C7446E89406B}";

        applicationObject = (_DTE)application;
        addInInstance = (AddIn)addInInst;
        windowToolWindow = applicationObject.Windows.
            CreateToolWindow (
                addInInstance, "VSUserControlHost.VSUserControlHostCtl",
```

```

        "Date Time Demo", guidstr, ref objTemp);

    windowToolWindow.Visible = true;
    objControl = (VSUserControlHostLib.IVSUserControlHostCtl)
        objTemp;
    System.Reflection.Assembly asm =
        System.Reflection.Assembly.GetExecutingAssembly();
    objControl.HostUserControl(asm.Location,
        "LocalizedAddin.LocalizedForm");
}

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

public void OnAddInsUpdate(ref System.Array custom)
{
}

public void OnStartupComplete(ref System.Array custom)
{
}

public void OnBeginShutdown(ref System.Array custom)
{
}

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

public void Exec(string commandName,
    EnvDTE.vsCommandExecOption executeOption,
    ref object varIn, ref object varOut, ref bool handled)
{

```

```

        handled = false;
        if(executeOption == EnvDTE.vsCommandExecOption.
            vsCommandExecOptionDoDefault)
        {
            if(commandName == "LocalizedAddin.Connect.Show")
            {
                windowToolWindow.Visible = true;
                handled = true;
                return;
            }
        }
    }
    private _DTE applicationObject;
    private AddIn addInInstance;
    public Window windowToolWindow;
    public VSUserControlHostLib.IVSUserControlHostCtl objControl;
}
}
}

```

Now create a User Control as I describe in the section titled “Using the Form Designer with a Tool Window” in Chapter 7. Put three labels on the form, as shown in Figure 8.5. Also add a time control. (You can find the time control in the Components section of the toolbox.)

Double-click the timer control and add the following code:

```

System.DateTime dt = System.DateTime.Now;
label2.Text = dt.ToLongDateString();
label3.Text = dt.ToLongTimeString();

```

Now return to the form designer. Set the top label’s Text property to `Date and Time`. Next set the form’s `Localizable` property to `True` and switch the language to `Spanish (Mexico)`. Now set the top label’s Text property to `Fecha y hora`. Next click back on the form, and in the Property window set the language to `French (France)`. Set the top label’s Text property to `Date et heure`. Now set the form’s language back to `Default`.

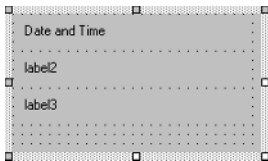


Figure 8.5 The form with three labels on it.

Here's the code for the form:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;

namespace LocalizedAddin
{
    /// <summary>
    /// Summary description for LocalizedForm.
    /// </summary>
    public class LocalizedForm : System.Windows.Forms.UserControl
    {
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.Label label3;
        private System.Timers.Timer timer1;
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;

        public LocalizedForm()
        {
            // This call is required by the Windows.Forms Form Designer.
            InitializeComponent();

            // TODO: Add any initialization after the InitForm call

        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if(components != null)
                {
                    components.Dispose();
                }
            }
        }
    }
}
```

```
    }
    base.Dispose( disposing );
}

#region Component Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    System.Resources.ResourceManager resources =
        new System.Resources.ResourceManager(
            typeof(LocalizedForm));
    this.label1 = new System.Windows.Forms.Label();
    this.label2 = new System.Windows.Forms.Label();
    this.label3 = new System.Windows.Forms.Label();
    this.timer1 = new System.Timers.Timer();
    ((System.ComponentModel.ISupportInitialize)(
        (this.timer1))).BeginInit();
    this.SuspendLayout();
    //
    // label1
    //
    this.label1.AccessibleDescription = ((string)(resources.
        GetObject("label1.AccessibleDescription")));
    this.label1.AccessibleName = ((string)(resources.
        GetObject("label1.AccessibleName")));
    this.label1.Anchor = ((System.Windows.Forms.AnchorStyles)
        (resources.GetObject("label1.Anchor")));
    this.label1.AutoSize = ((bool)(resources.
        GetObject("label1.AutoSize")));
    this.label1.Dock = ((System.Windows.Forms.DockStyle)
        (resources.GetObject("label1.Dock")));
    this.label1.Enabled = ((bool)(resources.
        GetObject("label1.Enabled")));
    this.label1.Font = ((System.Drawing.Font)(resources.
        GetObject("label1.Font")));
    this.label1.Image = ((System.Drawing.Image)(resources.
        GetObject("label1.Image")));
    this.label1.ImageAlign = ((System.Drawing.ContentAlignment)
        (resources.GetObject("label1.ImageAlign")));
    this.label1.ImageIndex = ((int)(resources.GetObject
        ("label1.ImageIndex")));
    this.label1.ImeMode = ((System.Windows.Forms.ImeMode)
        (resources.GetObject("label1.ImeMode")));
    this.label1.Location = ((System.Drawing.Point)
        (resources.GetObject("label1.Location")));
    this.label1.Name = "label1";
    this.label1.RightToLeft =
```

```

((System.Windows.Forms.RightToLeft)
    (resources.GetObject("label1.RightToLeft")));
this.label1.Size = ((System.Drawing.Size)(resources.
    GetObject("label1.Size")));
this.label1.TabIndex = ((int)(resources.
    GetObject("label1.TabIndex")));
this.label1.Text = resources.GetString("label1.Text");
this.label1.TextAlign = ((System.Drawing.ContentAlignment)
    (resources.GetObject("label1.TextAlign")));
this.label1.Visible = ((bool)(resources.
    GetObject("label1.Visible")));
//
// label2
//
this.label2.AccessibleDescription = ((string)(resources.
    GetObject("label2.AccessibleDescription")));
this.label2.AccessibleName = ((string)(resources.
    GetObject("label2.AccessibleName")));
this.label2.Anchor = ((System.Windows.Forms.AnchorStyles)
    (resources.GetObject("label2.Anchor")));
this.label2.AutoSize = ((bool)(resources.
    GetObject("label2.AutoSize")));
this.label2.Dock = ((System.Windows.Forms.DockStyle)
    (resources.GetObject("label2.Dock")));
this.label2.Enabled = ((bool)(resources.
    GetObject("label2.Enabled")));
this.label2.Font = ((System.Drawing.Font)(resources.
    GetObject("label2.Font")));
this.label2.Image = ((System.Drawing.Image)
    (resources.GetObject("label2.Image")));
this.label2.ImageAlign = ((System.Drawing.ContentAlignment)
    (resources.GetObject("label2.ImageAlign")));
this.label2.ImageIndex = ((int)(resources.GetObject
    ("label2.ImageIndex")));
this.label2.ImeMode = ((System.Windows.Forms.ImeMode)
    (resources.GetObject("label2.ImeMode")));
this.label2.Location = ((System.Drawing.Point)(resources.
    GetObject("label2.Location")));
this.label2.Name = "label2";
this.label2.RightToLeft =
((System.Windows.Forms.RightToLeft)
    (resources.GetObject("label2.RightToLeft")));
this.label2.Size = ((System.Drawing.Size)(resources.
    GetObject("label2.Size")));
this.label2.TabIndex = ((int)(resources.
    GetObject("label2.TabIndex")));
this.label2.Text = resources.GetString("label2.Text");
this.label2.TextAlign = ((System.Drawing.ContentAlignment)
    (resources.GetObject("label2.TextAlign")));
this.label2.Visible = ((bool)(resources.

```



```
        GetObject("label2.Visible"));
//
// label3
//
this.label3.AccessibleDescription = ((string)(resources.
    GetObject("label3.AccessibleDescription")));
this.label3.AccessibleName = ((string)(resources.
    GetObject("label3.AccessibleName")));
this.label3.Anchor = ((System.Windows.Forms.AnchorStyles)
    (resources.GetObject("label3.Anchor")));
this.label3.AutoSize = ((bool)(resources.
    GetObject("label3.AutoSize")));
this.label3.Dock = ((System.Windows.Forms.DockStyle)
    (resources.GetObject("label3.Dock")));
this.label3.Enabled = ((bool)(resources.
    GetObject("label3.Enabled")));
this.label3.Font = ((System.Drawing.Font)(resources.
    GetObject("label3.Font")));
this.label3.Image = ((System.Drawing.Image)
    (resources.GetObject("label3.Image")));
this.label3.ImageAlign = ((System.Drawing.ContentAlignment)
    (resources.GetObject("label3.ImageAlign")));
this.label3.ImageIndex = ((int)(resources.
    GetObject("label3.ImageIndex")));
this.label3.ImeMode = ((System.Windows.Forms.ImeMode)
    (resources.GetObject("label3.ImeMode")));
this.label3.Location = ((System.Drawing.Point)
    (resources.GetObject("label3.Location")));
this.label3.Name = "label3";
this.label3.RightToLeft =
((System.Windows.Forms.RightToLeft)
    (resources.GetObject("label3.RightToLeft")));
this.label3.Size = ((System.Drawing.Size)(resources.
    GetObject("label3.Size")));
this.label3.TabIndex = ((int)(resources.
    GetObject("label3.TabIndex")));
this.label3.Text = resources.GetString("label3.Text");
this.label3.TextAlign = ((System.Drawing.
    ContentAlignment)(resources.
    GetObject("label3.TextAlign")));
this.label3.Visible = ((bool)(resources.
    GetObject("label3.Visible")));
//
// timer1
//
this.timer1.Enabled = true;
this.timer1.SynchronizingObject = this;
this.timer1.Elapsed += new System.Timers.
    ElapsedEventHandler(this.timer1_Elapsed);
//
```

```

// LocalizedForm
//
this.AccessibleDescription = ((string)(resources.
    GetObject("$this.AccessibleDescription")));
this.AccessibleName = ((string)(resources.GetObject
    ("$this.AccessibleName")));
this.Anchor = ((System.Windows.Forms.AnchorStyles)
    (resources.GetObject("$this.Anchor")));
this.AutoScroll = ((bool)(resources.
    GetObject("$this.AutoScroll")));
this.AutoScrollMargin = ((System.Drawing.Size)
    (resources.GetObject("$this.AutoScrollMargin")));
this.AutoScrollMinSize = ((System.Drawing.Size)
    (resources.GetObject("$this.AutoScrollMinSize")));
this.BackgroundImage = ((System.Drawing.Image)
    (resources.GetObject("$this.BackgroundImage")));
this.Controls.AddRange(new System.Windows.Forms.Control[] {
    this.label3,
    this.label2,
    this.label1});
this.Dock = ((System.Windows.Forms.DockStyle)(resources.
    GetObject("$this.Dock")));
this.Enabled =
((bool)(resources.GetObject("$this.Enabled")));
this.Font = ((System.Drawing.Font)(resources.
    GetObject("$this.Font")));
this.ImeMode = ((System.Windows.Forms.ImeMode)
    (resources.GetObject("$this.ImeMode")));
this.Location = ((System.Drawing.Point)(resources.
    GetObject("$this.Location")));
this.Name = "LocalizedForm";
this.RightToLeft = ((System.Windows.Forms.RightToLeft)
    (resources.GetObject("$this.RightToLeft")));
this.Size = ((System.Drawing.Size)(resources.
    GetObject("$this.Size")));
this.TabIndex = ((int)(resources.
    GetObject("$this.TabIndex")));
this.Visible = ((bool)(resources.
    GetObject("$this.Visible")));
((System.ComponentModel.ISupportInitialize)(resources.
    GetObject("$this.Visible"))).EndInit();
this.timer1.EndInit();
this.ResumeLayout(false);

}
#endregion

private void timer1_Elapsed(object sender,
    System.Timers.ElapsedEventArgs e)
{
    System.DateTime dt = System.DateTime.Now;

```

```
        label2.Text = dt.ToLongDateString();  
        label3.Text = dt.ToLongTimeString();  
    }  
}
```

When you run this add-in, you will see the form for it. By default the form will be in English. If you switch your computer's culture to either French (France) or Spanish (Mexico), and then restart the IDE, your add-in will be in the appropriate language. Notice also that the date and time will use the format standard of the particular culture.

Moving Forward

In this chapter I discussed three important topics: the life cycle of an add-in, how to debug an add-in, and how to use satellite DLLs to allow your software to present information in a local language. Globalization is something that many programmers tend to neglect (especially in the United States), but if you add globalization features to your programs, your software will be much more well received by people in other countries.

In the next chapter, I explain how you can manipulate solutions and project programmatically from either an add-in or a macro. Stay tuned!

Manipulating Solutions and Projects

In Chapter 5, “Just Enough .NET Architecture,” I talked briefly about project objects and how you can access project information using different objects, depending on the language the project is written in. I expand on that discussion here, by explaining how you can modify the project information.



If you have not read Chapter 5, I encourage you to read the section titled “The Visual Studio Project Types” there before continuing with this chapter, because I assume here that you understand how to access generic project information through the Project object, and that this Project object also contains an Object property. The Object property gives you access to a COM object that contains language-specific information.

Why would you want to modify project information? Suppose you are building a project in which you call several external tools during the build process. To access these external tools, you set various Custom Build steps in the properties for the particular files in your project. You could set the Custom Build step for each file separately or you could write a macro that sets the information for you. This, in fact, is exactly what I did in Chapter 8 in the macro that adds resource files to a project.

As another example, you might be developing a class library contained in an assembly for commercial use by other programmers. In order to make things as easy on your clients as possible (and to minimize support calls!) you might include with your library a macro or add-in that automatically adds the assembly to a project in the form of a

reference, carefully setting the project properties. Then the user can begin programming with your class library without having to spend time messing with the project settings. You could include such a macro with your assembly.

Determining the Currently Selected Project

The following macro code obtains the currently selected project in the Solution Explorer. When you have a macro that manipulates a project, using this code, you can find out which project name the IDE user has clicked in the Solution Explorer. The IDE user can also click on one of the items inside a project, and this code will obtain the project containing the selected item.

```
Sub FindProject()  
    Dim projs As System.Array  
    Dim proj As Project  
    projs = DTE.ActiveSolutionProjects()  
    If projs.Length > 0 Then  
        proj = projs.GetValue(0)  
        MsgBox(proj.Name)  
    End If  
End Sub
```

This macro first obtains the root DTE object, and from there a list of the active projects through the `ActiveSolutionProjects` property. (Note: The reason that “projects” is plural here is because the IDE user can click on multiple items in the Solution Explorer by clicking one item and then holding down the Ctrl key and clicking another item.) The preceding macro obtains only the first selected project in such cases. If you want your macro to operate on all the selected projects, you can use this macro instead:

```
Sub FindProjects()  
    Dim projs As System.Array  
    Dim proj As Project  
    projs = DTE.ActiveSolutionProjects()  
    If projs.Length > 0 Then  
        For Each proj In projs  
            MsgBox(proj.Name)  
        Next  
    End If  
End Sub
```

You can use similar code in an add-in as well. Remember that in add-ins, you do not access the root DTE object through the DTE variable name; instead, you grab the DTE object from the first parameter in the `OnConnection` method. Here’s the `OnConnection` method for a VB.NET add-in that registers a command that will get the selected projects. (If you want to try this out, make sure you check the Add-in wizard option to create a Tool menu item; that will give you a class that’s derived from `IDTCommandTarget` so you can implement commands.)

```

Public Sub OnConnection(ByVal application As Object, _
    ByVal connectMode As Extensibility.ext_ConnectMode, _
    ByVal addInInst As Object, ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnConnection

    applicationObject = CType(application, EnvDTE.DTE)
    addInInstance = CType(addInInst, EnvDTE.AddIn)
    Dim objAddIn As AddIn = CType(addInInst, AddIn)
    Dim CommandObj As Command
    Try
        CommandObj = applicationObject.Commands.AddNamedCommand( _
            objAddIn, "GetProjects", "AddinProjectManip2", _
            "Gets the selected project names ", True, 59, Nothing, _
            1 + 2)
    Catch e As System.Exception
    End Try
End Sub

```

Notice that I'm adding a command called `GetProjects`. Here's the `QueryStatus` method that enables the command:

```

Public Sub QueryStatus(ByVal cmdName As String, ByVal neededText _
    As vsCommandStatusTextWanted, ByRef statusOption As vsCommandStatus, _
    ByRef commandText As Object) Implements IDTCommandTarget.QueryStatus
    If neededText = EnvDTE.vsCommandStatusTextWanted. _
        vsCommandStatusTextWantedNone Then
        If cmdName = "AddinProjectManip2.Connect.GetProjects" Then
            statusOption = CType(vsCommandStatus.vsCommandStatusEnabled _
                + vsCommandStatus.vsCommandStatusSupported, vsCommandStatus)
        Else
            statusOption = vsCommandStatus.vsCommandStatusUnsupported
        End If
    End If
End Sub

```

And, finally, here's the code that executes the command. You can see that I just pasted in the macro code (that's why I chose Visual Basic for this add-in) and then replaced `DTE` with `applicationObject`.

```

Public Sub Exec(ByVal cmdName As String, ByVal executeOption As _
    vsCommandExecOption, ByRef varIn As Object, ByRef varOut As Object, _
    ByRef handled As Boolean) Implements IDTCommandTarget.Exec
    handled = False
    If (executeOption = vsCommandExecOption. _
        vsCommandExecOptionDoDefault) Then
        If cmdName = "AddinProjectManip2.Connect.GetProjects" Then
            Dim projs As System.Array
            Dim proj As Project
            projs = applicationObject.ActiveSolutionProjects()

```

```

        If projs.Length > 0 Then
            For Each proj In projs
                MsgBox(proj.Name)
            Next
        End If
        handled = True
        Exit Sub
    End If
End If
End Sub

```

To try out this add-in, build its project, start a new instance of Visual Studio .NET, and open a solution (any solution will do). In the Solution Explorer, select a couple projects by clicking one, then while holding down the Ctrl key, clicking another. Next, open the Add-in Manager and check the box next to the add-in. Then choose View⇨Other Windows⇨Command Window, to open a new command window, and type the following command into the command window:

```
AddinProjectManip2.Connect.GetProjects
```

You will see a series of message boxes open, one for each project you selected, with each message box showing the name of a project.

Manipulating a Project's Items

By itself, the macros and add-ins in the preceding section aren't particularly useful in that they only display information about a project, rather than manipulate the projects. But you can easily add code to manipulate a Project object. Here are some of the project-related objects that you might manipulate from a macro.

Project.ProjectItems. Your macro could check whether a file is already part of a project, by checking for the file's existence in the `ProjectItems` collection. If the file doesn't exist, your macro could add it. For example, if you have a class library in the form of source code, your macro could automatically add the source code to the project. You can also use the `ProjectItems` property to obtain information on the individual items in the project, such as the source code files or the resource files. Each such item is a `ProjectItem` object.



Think of the `ProjectItems` object as corresponding to the items underneath the project name in the Solution Explorer. Remember, the project has a `ProjectItem` object not just for files, but for folders as well. If, for example, you have a C++ project with folders called Source Files, Header Files, and Resource Files, you will have a separate `ProjectItem` object for the three folders as well as for each file. However, in the case of C# and VB.NET projects, you will *not* have a `ProjectItem` object for the References folder, nor its members.

ProjectItem.IsOpen. Your macro can check this property to determine if the user currently has the file open in the IDE. For example, if the `ProjectItem` object corresponds to a C++ source file, then `IsOpen` will be true if the C++ source file is currently open in the IDE editor. If your macro or add-in is making considerable changes to a project, you might check `IsOpen` for each item in the project. If any such items are open, you might display a message to the IDE user stating that the documents must be closed before proceeding. (Note: The `IsOpen` method works only for items in VB.NET and C# projects.) The following code is an example of a macro that uses `IsOpen`. This macro goes through the list of `ProjectItem` objects and determines which are opened, finally displaying a message box showing the list of open project items.

```
Sub ListOpenItems()
    Dim projs As System.Array
    Dim proj As Project
    Dim pitem As ProjectItem
    projs = DTE.ActiveSolutionProjects()
    If projs.Length > 0 Then
        proj = projs.GetValue(0)
        Dim str As String = ""
        For Each pitem In proj.ProjectItems
            If pitem.IsOpen Then
                str &= pitem.Name + Chr(13) + Chr(10)
            End If
        Next
        MsgBox(str)
    End If
End Sub
```

ProjectItem.Saved. Your macro can check if a project item has been changed since the last save. For example, if the IDE user has a C++ source file open in the IDE editor, and the user edits the source code but does not save the file, then the `Saved` property for the corresponding project item will be false.

ProjectItem.Open. Your macro can open a project item automatically. If the project item is a source file, the file will open in the editor. Your macro can then make changes to the source file. Note, however, that there's a trick to making the `Open` function work: The `Open` function returns an object of class `Window`, and initially this `Window` object's `Visible` property is set to `False`. You need to change the `Visible` property to `True`. The following code demonstrates this:

```
Sub OpenAllSourceFiles()
    Dim projs As System.Array
    Dim proj As Project
    Dim pitem As ProjectItem
    projs = DTE.ActiveSolutionProjects()
    If projs.Length > 0 Then
        proj = projs.GetValue(0)
        Dim ext As String = ""
        For Each pitem In proj.ProjectItems
            ext = System.IO.Path.GetExtension(pitem.Name)
        Next
    End If
End Sub
```



```
    If ext = ".cpp" Or ext = ".vb" Or ext = ".cs" Then
        Dim win As Window
        win = pitem.Open()
        win.Visible = True
    End If
Next
End If
End Sub
```

ProjectItem.Remove and ProjectItem.Delete. Be sure you understand the difference between these two. The `Remove` method removes the item from the project, but keeps the item on the disk. If, for example, the item is a source file, after you call the `Remove` method, the source file will still exist on the disk, but will no longer be a member of the project. The `Delete` method, in contrast, removes the item from the project and *deletes* the file from the disk, so use `Delete` with care.

ProjectItem.Save. If the project item is open in the editor, and has changed since the last save, this method will save the item. However, the `Save` method works only for items in C# and VB.NET projects.

ProjectItem.SaveAs. If the project item is open in the editor, you can use `SaveAs` to save the project item under a different name. The `SaveAs` method will save the project item with the new name, remove the original file from the project (leaving the original project item's file on the disk), and add the new file to the project. However, `SaveAs` has two caveats: First, you can use it only on files that are currently open in the editor, which means the source code editor for source files or a resource editor for resource files; second, `SaveAs` works only for items in a C# or VB.NET project.

ProjectItem.Name. The `Name` property represents the name of the project item. For folders (such as in a C++ project), this is the name of the folder. For files, this name always matches the filename. You can change this property, in the case of folders to change the name of the folder. However, if you change this property for a file, nothing will happen: neither the name of the file will change, nor will the filename in the project tree in the Solution Explorer. If you need to change the name of a file, see `SaveAs`, just defined.

ProjectItem.FileCount. In the case of `ProjectItem` objects that are folders (such as in a C++ project), the `FileCount` property will tell you how many items are in the folder. For individual files, this property is always 1.

ProjectItem.FileNames. Be careful with this property, as it does not behave as the documentation states it will. This is an array that is supposed to contain the list of filenames in the project item. In the case of project items that are a single file, the `FileNames` array contains only a single item, which is a string representing the full path and filename of the single file. So far so good; but in the case of folders, the `FileNames` property doesn't quite function as you would expect. For the version of Visual Studio .NET that is current at the time of this writing (the first version), all the items in the `FileNames` array contain the name of the folder itself, *not* the files contained in the folder.

In addition to the preceding items, the `ProjectItems` property of the `Project` object also has several methods that let you add items to a project. The `ProjectItem` object's methods, for example, let you write a macro that automatically adds a class library to a project. You can also add folders to the project. The `ProjectItem` object's methods are rather intelligent, in that they take into consideration the fact that you probably don't want to add a file to a project that is not within the project's directory structure.

To add a file to a project, you have several choices. If you want to add a single file to the project, you can use the `AddFromFile` method. Here's an example:

```
Sub AddSingleFile()
    Dim projs As System.Array
    Dim proj As Project
    Dim pitems As ProjectItems
    projs = DTE.ActiveSolutionProjects()
    If projs.Length > 0 Then
        proj = projs.GetValue(0)
        pitems = proj.ProjectItems
        pitems.AddFromFile("c:\temp\myfile.bas")
    End If
End Sub
```

You can see that the call to `AddFromFile` adds the file called `c:\temp\myfile.bas`.

When the IDE adds the file, it will assume a build action based on the filename extension. Since this code adds a file with a `.bas` extension, if you run this macro on a VB.NET project, the `myfile.bas` will be set to build with the project. However, if you add a resource file, such as a resource, its Build Action is set, by default, to Content. You can change this to Embedded Resource.

The problem with this code, however, is that if your project is not in the temp directory, you will have a file in the project that's not inside the project's directory. Further, the file's name has an absolute path in it, `c:\temp`, which can cause further trouble if you want to copy the project onto another computer.



Normally, if you remove a file from a VB.NET or C# project by right-clicking the filename in the Solution Explorer, and in the popup menu choosing Delete, you will be warned that, “myfile.vb’ will be deleted permanently.” If you click OK, the file itself will be deleted. However, this message appears only when the file is in the project directory. If you use `AddFromFile` to add a file that's outside your project directory, and you delete the file from the project, Visual Studio .NET will not delete the file itself, nor will it show a message box saying it plans to do so. (If you use `AddFromFile` to add a file that's in the project's directory and you try to delete the file from the project, then the IDE will delete the file itself.)

Fortunately, the `ProjectItems` object includes another method, called `AddFromFileCopy`, that copies the file into the project directory, then adds the copy—not the original—to the project. Here's an example:

```

Sub AddSingleFileCopy()
    Dim projs As System.Array
    Dim proj As Project
    Dim pitems As ProjectItems
    projs = DTE.ActiveSolutionProjects()
    If projs.Length > 0 Then
        proj = projs.GetValue(0)
        pitems = proj.ProjectItems
        pitems.AddFromFileCopy("c:\temp\myfile.bas")
    End If
End Sub

```

Of course, the `AddFromFileCopy` also has a disadvantage: You now have a second copy of the original file. This means you have to decide whether you would prefer to use `AddFromFile` or `AddFromFileCopy`.



The `AddFromFile` and `AddFromFileCopy` functions both return an instance of `ProjectItem`, which represents the item you just added to the project. You can then manipulate the new item through the returned `ProjectItem` object.

When you call `AddFromFile`, if the file you're adding already exists, then you will get an error. In the case of the macros, you will see a message box appear with the message "There is already a link to 'c:\temp\myfile.bas'. A project cannot have more than one link to the same file." If you prefer, you can handle the error yourself using a `Try/Catch` block; doing so will suppress the default error message. Here's an example:

```

Try
    pitems.AddFromFile("c:\temp\myfile.bas")
Catch
    MsgBox("The project already has a myfile.bas")
End Try

```

You will also get an error if you try to add a file that has the same name as a file already in the project. For example, if your project has a file called `myfile.bas`, and you add a *different* file also called `myfile.bas`, you will get the message, "There is already a file of the same name in this folder."

If you try to add a file that simply doesn't exist, you will get a different error message in a message box that reads: "Cannot add the link because the source file 'c:\myfile.bas' cannot be found." As before, you can handle this error with a `Try/Catch` block if you prefer.

Finally, in an attempt to exhaust all possibilities, I explored what would happen if I called `AddFromFileCopy`, passing a full-path to a file that's in the project directory. It turns out the IDE doesn't attempt to copy the file (which, I suppose, would result in an error); instead, the IDE just adds the file itself to the project, meaning you are not working with a copy of the file, but the original. Therefore, if you try to delete the file from the project, you will delete the original file itself. So be careful when doing this.

If you want to add an entire directory of files to a project, you can use the `AddFromDirectory` method. Use care when calling `AddFromDirectory`, because you could end up with files you didn't expect: Even subdirectories and their contents will get added to your project. If the files don't have any business being in a project, then the compiler won't know what to do with them and will simply ignore them when you build the project. If the files do, however, belong, then the IDE will build any source files when you perform a build. However, as with adding a single file for C# and VB.NET projects, resource files (such as .jpg files) will not, by default be set to Embedded Resource for their Build Action; instead, the Build Action for resource files is set by default to Content.

When you call `AddFromDirectory`, you will end up with another folder in your project that contains links to all the files in the directory. The folder will have the same name as the directory. Here's a macro that adds an entire directory to the currently selected project:

```
Sub AddEntireDirectory()  
    Dim projs As System.Array  
    Dim proj As Project  
    Dim pitems As ProjectItems  
    projs = DTE.ActiveSolutionProjects()  
    If projs.Length > 0 Then  
        proj = projs.GetValue(0)  
        pitems = proj.ProjectItems  
        pitems.AddFromDirectory("c:\temp2")  
    End If  
End Sub
```

Interestingly, if you then delete the folder from your project, you will receive a warning message that the entire directory and its contents will be deleted. I tried this, and when I clicked OK, to my surprise the items were indeed removed from the project; but the original files were still on my hard drive, so, in fact, the files were *not* permanently deleted. It's hard to know whether this is a bug or a feature, but in case it's a bug, I wouldn't count on the files being there in future releases of Visual Studio .NET.

Manipulating a Project's Settings

In Chapter 5, "Just Enough .NET Architecture," I talked briefly about language-specific configurations. Here I'm going to expand on that discussion by talking about general configurations, as well as how to manipulate both general configurations and language-specific configurations.

When adding a library to a project, there's an alternative to simply adding the library's code, which helps avoid having either an absolute path or a copy of the file. In the case of a VB.NET or C# project, you could add a reference to your library, rather than actually adding your library's code files to the project. But that, of course, means your library must exist as an assembly. And in the case of a C++ project, you can add the library's .lib file to the project's linker section. To do either of these tasks, you need to work with the project settings.

When you work on a project and you right-click the project's name in the Solution Explorer and choose Properties, you will see the Property Pages dialog box, which allows you to modify the project settings. But in addition to the project settings, you can manipulate build settings for individual items in the project. Through the Configuration object you can manipulate the settings for either a project or the project items. For a project, you access the configurations through the `Project.ConfigurationManager` property; for a project's item, you access the configurations through the `ProjectItem.ConfigurationManager` property.



Although the `ProjectItem` object is language-independent, not all languages have a `ConfigurationManager` for the project items. C# and VB.NET do not; C++ does.

Normally, a single project has at least two configurations, one for Debug and one for Release. Thus, a project's `ConfigurationManager` property will be an array containing at least two items. Each item is of class `Configuration`.

The following macro obtains the `Configuration` objects for a project and each of its items:

```
Sub GetConfigs()
    Dim projs As System.Array
    Dim proj As Project
    Dim pitem As ProjectItem
    Dim pitems As ProjectItems
    projs = DTE.ActiveSolutionProjects()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    If projs.Length > 0 Then
        proj = projs.GetValue(0)
        Dim cfg As Configuration
        VBMacroUtilities.Print("Project Configurations: " & _
            proj.ConfigurationManager.Count)
        For Each cfg In proj.ConfigurationManager
            VBMacroUtilities.Print("    " & cfg.ConfigurationName)
        Next
        For Each pitem In proj.ProjectItems
            VBMacroUtilities.Print("Item " & pitem.Name)
            If Not pitem.ConfigurationManager Is Nothing Then
                For Each cfg In pitem.ConfigurationManager
                    VBMacroUtilities.Print("    " &
                        cfg.ConfigurationName)
                Next
            End If
        Next
    End If
End Sub
```

When you run this macro for a project, you will see the names of the different configurations available. Notice in this code that I'm stepping through the list of configurations

using the `For Each` construct. If, however, you know that a certain configuration is available, you can access the configuration directly by name, like so:

```
cfg = proj.ConfigurationManager.Item("Debug", "Win32")
```

This line assumes `proj` is a `Project` object, and `cfg` is a `Configuration` object. The first parameter is the configuration name; the second parameter is the platform for the configuration. This line of code, then, also assumes the project is a C++ project and that it has a Debug configuration that runs on Win32.

Accessing and Setting Configuration Properties

When you open up the Property Pages dialog box for a project, you can set the different properties for the various configurations. The `Configuration` object gives you access to these different properties through the `Configuration.Properties` object. For each property in the Property Pages dialog, the `Properties` object contains a single instance of class `Property`. This `Property` object contains a key and a value. The key is the name of the property, and the value is the property's value.

For example, when you have a C++ project and you open the Property Pages dialog box for this project, under the Debugging setting you will find a property called "Working Directory". This is the setting for the directory under which your program should run when you are debugging the program. As with all the settings in the Property Pages dialog box, this Working Directory setting has a corresponding instance of `Property` that contains a name and a value. The name, in this case, happens to be "WorkingDirectory", and the value is a string that is stored in the working directory, if any. (If you leave the Working Directory setting blank, which it is by default, the `Value` member of the `Property` object will be set to `Nothing` in VB.NET, which corresponds to `NULL` in C++.) Since each item in the Property Pages dialog box is a single property, you can see why, in Visual Studio .NET, Microsoft named the project settings dialog box the Property Pages dialog box.

Here's a macro that will list all the properties for a project:

```
Sub ConfigurationProperties()
    Dim projs As System.Array
    Dim proj As Project
    Dim pitem As ProjectItem
    Dim pitems As ProjectItems
    projs = DTE.ActiveSolutionProjects()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    If projs.Length > 0 Then
        proj = projs.GetValue(0)
        Dim cfg As Configuration
        VBMacroUtilities.Print("Project Configurations: " & _
            proj.ConfigurationManager.Count)
        For Each cfg In proj.ConfigurationManager
```

```

VBMacroUtilities.Print(cfg.ConfigurationName)
Dim prop As EnvDTE.Property
' Or use: Dim prop As [Property]
For Each prop In cfg.Properties
    If Not prop.Value Is Nothing Then
        VBMacroUtilities.Print("  " & prop.Name & _
            ": " & prop.Value.ToString())
    Else
        VBMacroUtilities.Print("  " & prop.Name & _
            ": <None>")
    End If
Next
End If
End Sub

```

I want to point out something strange about this code, specifically related to the comment that reads, `Or use: Dim prop as [Property]`. It means that you could use that line instead of the previous line, `Dim prop as EnvDTE.Property`. The square brackets are used to distinguish the type name from the built-in VB.NET keyword `property`. However, because I prefer to avoid resorting to odd syntax, I simply fully qualify the word `Property` by preceding it with `EnvDTE`, which is the namespace where you can find the `Property` class.

When you click a project in the Solution Explorer and then run this macro, it will step through each configuration; and then for each configuration, it will step through all the properties, listing the name and value of each. This is a pretty useful macro if later on you're going to write another macro that modifies the properties, because you can look at this macro and figure out the names of the properties. (This macro is, in fact, the one I used to help me figure out that the property name for the working directory in a C++ program is "WorkingDirectory".)

Once you know the name of a property, you can change it. Here, then, is a macro that sets a single property, in this case, the `WorkingDirectory` property:

```

Sub SetSingleProperty()
    Dim projs As System.Array
    Dim proj As Project
    Dim path As String
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    projs = DTE.ActiveSolutionProjects()
    If projs.Length > 0 Then
        proj = projs.GetValue(0)
        If proj.Kind <> "{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}" Then
            MsgBox("Please select a C++ project.")
            Exit Sub
        End If
    Try
        Dim cfg As Configuration
        cfg = proj.ConfigurationManager.Item("Debug", "Win32")
        Dim prop As EnvDTE.Property
    
```

```

prop = cfg.Properties.Item("WorkingDirectory")
If Not prop.Value Is Nothing Then
    VBMacroUtilities.Print(prop.Value)
Else
    VBMacroUtilities.Print("<No value specified>")
End If
prop.Value = "c:\temp"
Catch
    MsgBox("Exception caught.")
End Try
End If
End Sub

```

The first key line in this code is where I obtain the property itself by accessing the `Item` member of the `Properties` object, passing the name "WorkingDirectory". That process gives me back a `Property` instance. The second key line is where I change the value of the actual property; specifically, the `Property` instance's `Value` member. In this case, I set the value to "c:\temp".

Notice also that to access the particular configuration, I specified the name "Debug" and the platform, "Win32". And notice that I wrapped the configuration code inside a `Try/Catch` block; that way, if I made a mistake when I typed in the code, such as typing the name of the property wrong, I would catch the error in the `Catch` block.

If you click on a C++ project in the Solution Explorer and run this `SetSingleProperty` macro, then open the Property Pages for the project and click on the Debugging group, you'll see that the working directory is now set to c:\temp. (If this were an important project, you might want to set it back to what it used to be. To help you out, I wrote the macro so it would print out the previous value to the Output window.)

Now when I run the earlier macro, called `ConfigurationProperties`, on a VB.NET project, I can see that the property for setting the working directory is instead called `StartWorkingDirectory`. Thus, if you want to modify the `SetSingleProperty` macro to set the working directory for a VB.NET program, you can first change the if-block that checks the project `Kind` property, like so:

```

If proj.Kind <> VSLangProj.PrjKind.prjKindVBProject Then
    MsgBox("Please select a VB project.")
Exit Sub
End If

```

Then, you can change the configuration line to this:

```

cfg = proj.ConfigurationManager.Item("Debug", ".NET")

```

Notice that here you specify `.NET` for the platform, not `Win32`.

Then you can change the line that retrieves the property, like so:

```

prop = cfg.Properties.Item("StartWorkingDirectory")

```

Finally, if you want to use this macro on C# projects, the property name is the same, `StartWorkingDirectory`.

Adding Configurations

In Chapter 8, I showed you a trick that will let you test an add-in either by running Visual Studio .NET in command-line mode or in standard GUI mode. Recall that this trick involved adding a new configuration specifically for running and debugging in command-line mode. This new configuration set various properties for launching the external program. These properties were Start External Program, Command-Line Arguments, and Working Directory.

Using the language-specific configuration objects, you can modify the project using a macro to make these same settings. Here are the steps involved:

1. Add a new configuration based on an existing configuration.
2. Set the property for starting an external program.
3. Set the property for the command-line arguments.
4. Set the property for the working directory.

To add a new configuration (step 1), you use the language-independent `ConfigurationManager` object. To set the properties (steps 2 through 4), you can use the information in the previous section, "Accessing and Setting Configuration Properties." Following is a macro that does this work. The assumption here is that you will want to load Visual Studio .NET in command-line mode to build some other solution, during which you want to test out your add-in. (This solution should not contain your add-in; it would contain separate projects you are working on.)

```
Sub AddCommandLineConfiguration()
    Dim projs As System.Array
    Dim proj As Project
    Dim path As String
    projs = DTE.ActiveSolutionProjects()
    If projs.Length > 0 Then
        proj = CType(projs.GetValue(0), EnvDTE.Project)
        If proj.Kind <> VSLangProj.PrjKind.prjKindVBProject And _
            proj.Kind <> VSLangProj.PrjKind.prjKindCSharpProject Then
            MsgBox("Please select a VB or C# project.")
            Exit Sub
        End If

        ' Get the build solution name and path
        Dim buildsoln As String
        Dim solndir As String
        buildsoln = InputBox( _
            "Enter the path and filename of the solution.")
        If buildsoln = "" Then
            Exit Sub
        End If
        solndir = System.IO.Path.GetDirectoryName(buildsoln)
        buildsoln = System.IO.Path.GetFileName(buildsoln)
        ' Get the installed path
```

```

Dim reg As Microsoft.Win32.RegistryKey
Dim installed As String
reg = Microsoft.Win32.Registry.LocalMachine.OpenSubKey( _
    "SOFTWARE\Microsoft\VisualStudio\7.0")
installed = reg.GetValue("InstallDir")

' Add a configuration
proj.ConfigurationManager.AddConfigurationRow( _
    "CmdDebug", "Debug", True)

' Now locate the configuration we just added
Try
    Dim command As String = installed & "devenv.exe"
    Dim args As String = buildsoln & " /build DEBUG"
    Dim cfg As Configuration
    Dim prop As EnvDTE.Property

    cfg = proj.ConfigurationManager.Item("CmdDebug", ".NET")
    prop = cfg.Properties.Item("StartProgram")
    prop.Value = command

    prop = cfg.Properties.Item("StartArguments")
    prop.Value = args

    prop = cfg.Properties.Item("StartWorkingDirectory")
    prop.Value = solndir

    prop = cfg.Properties.Item("StartAction")
    prop.Value = 1
Catch
    MsgBox("Exception caught.")
End Try
End If
End Sub

```

Chapter 8 explained the purpose of this code. This macro simply sets up the project the way you did manually in Chapter 8. In addition, notice that I look up the installation path for Visual Studio .NET. The reason is that the command line, when it runs `devenv.exe` (which is the executable file for Visual Studio .NET) needs a full path to the command being run. Thus, instead of simply putting `devenv.exe` for the command line, I extract the installation path from the Registry, which is also the directory of the `devenv.exe` program. Then I use this information to construct the full path and filename for the `devenv.exe` program, which I then store in the `StartProgram` property.

I also added one extra item in this code that might seem foreign: I set a property called `StartAction`. This corresponds to the radio button in the Start Action group in the Property Pages dialog box. The choices in the dialog box are Start Project, Start External Program, and Start URL. To choose one of these items programmatically, set the `StartAction` property to 0, 1, or 2, respectively. Since here I want to start an external program, I set the `StartAction` property to 1.

To use this macro, click on an add-in project (since the macro really only makes sense for add-in projects) and then run the macro. You will be prompted for the full path and filename of a solution file. This is the solution file that devenv will build in command-line mode while loading the add-in. The macro will then finish up, after which your add-in project will be set properly; from there you can proceed as you did in Chapter 8.

Configuring Projects at the Solution Level

Like projects, a solution also has properties that live as members of the Solution object, as well as key-value pairs in the Properties member. You can manipulate a solution through both of these.

Here's a macro that lists the pairs in the Properties member of the current Solution object:

```
Sub SolutionProperties()  
    Dim prop As EnvDTE.Property  
    VBMacroUtilities.Setup(DTE)  
    VBMacroUtilities.Clear()  
    For Each prop In DTE.Solution.Properties  
        Try  
            VBMacroUtilities.Print(prop.Name & _  
                ": " & prop.Value.ToString())  
        Catch  
            VBMacroUtilities.Print(prop.Name & _  
                ": <None>")  
        End Try  
    Next  
End Sub
```

When you run this macro, you will see StartupProject, one of the more useful properties from a macro perspective; by changing this property, you can set the startup project. Here's an example:

```
Sub SetStartupProperty()  
    Dim prop As EnvDTE.Property  
    prop = DTE.Solution.Properties.Item("StartupProject")  
    prop.Value = "TestCSharpProject"  
End Sub
```

In my solution, I have a project called TestCSharpProject; when I run this macro, that project becomes the new startup project. If you want to try out this macro, replace the string with a project in your solution.

Using the Solution object, you can also add and remove projects. To add a project, you start with a template. (The templates are sprinkled throughout the Visual Studio .NET installation directory.) Here's an example of a macro that creates a new Visual Basic Windows Application and adds it to the current solution:

```

Sub CreateVBApp()
    Dim apath As String
    apath = System.IO.Path.GetDirectoryName( _
        DTE.Solution.FullName) & "\MyVBProj2"
    DTE.Solution.AddFromTemplate( _
        "C:\Program Files\Microsoft Visual Studio .NET\" & _
        "Vb7\VBProjects\windowsapplication.vsz", _
        apath, "MyVBProj2", False)
End Sub
    
```

Note that, to accommodate space restrictions in this book, I broke up the path to the template into multiple lines; you can see that the first parameter is actually one long string representing the path. The second parameter to `AddFromTemplate` is the full path of where to put the project; I started with the solution's own path and added my own subdirectory, storing the string in the `apath` variable. The third parameter is the name of the project to be created. The final parameter is a Boolean value representing whether to create a new solution for the new project. I chose `False`, meaning I want to add the project to the current solution.



In the preceding code, notice that the file I used for the first parameter is a .vsz file. The online help has an error in it, stating that you should use a project file. However, the correct way to use the AddFromTemplate function is to use the .vsz wizard file. (For more information on .vsz files and wizards, see Chapter 12, "Creating Project Wizards.")

To remove a project from the solution, you need a reference to the project's `Project` object. Then you simply call `DTE.Solution.Remove(proj)`, where `proj` is the `Project` object. Next, you can either remove the files from the project's folder, in which case the project will be permanently removed from the hard drive, or you can leave the files in the project's folder, in which case trying to re-create the project by calling the `CreateVBApp` macro will fail, since the folder already has a project in it. (The solution to calling `CreateVBApp` again would be to give the `CreateVBApp` macro a new project name and project folder.)

Finally, here's a short macro that creates a brand-new empty solution into which you can add projects:

```

Sub CreateEmptySolution()
    DTE.Solution.Create("c:\dev", "MySolution.sln")
End Sub
    
```

When you run this macro, you will see a new solution in the main IDE, but note that you will *not* see the solution on the disk *until* you choose `File` → `Save All`.

Configuring Individual Files

In a C++ project, you can right-click on an individual file in the project and choose `Properties`; from there you can set properties for the specific file that are exceptions to

the usual project properties. Say that you have turned on optimization for the project, but want to exclude a single C++ file from being optimized. To do so, set the Optimization property to disabled for only that file, while leaving the property Enabled and set to, for example, Maximum Speed for the rest of the project. You would then set the optimization property to Maximum Speed property for the entire project, while setting the same property to Disabled for the individual file.

You can also set other properties, such as Exclude, from Build. During testing, there may be times when you want to periodically exclude a particular set of files from the build. While you could set a separate configuration for this purpose, you could also write a macro to exclude the files and another macro to include the files.

All that said, be aware of this confusing element in Visual Studio .NET regarding the configuration of individual files: You can find a `Properties` member for both a `ProjectItem` and a `ProjectItem` object's `Configuration` objects. Which do you use, and when? It depends on the language. Take a look at this macro, then try running it for various projects:

```
Sub GetProjectItemProps()
    Dim projs As System.Array
    Dim proj As Project
    Dim pitem As ProjectItem
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    projs = DTE.ActiveSolutionProjects()
    If projs.Length > 0 Then
        proj = projs.GetValue(0)
        For Each pitem In proj.ProjectItems
            VBMacroUtilities.Print(pitem.Name)
            Dim prop As EnvDTE.Property
            For Each prop In pitem.Properties
                Try
                    If Not prop.Value Is Nothing Then
                        VBMacroUtilities.Print("  " & prop.Name & _
                            ": " & prop.Value.ToString())
                    Else
                        VBMacroUtilities.Print("  " & prop.Name & _
                            ": <None>")
                    End If
                Catch
                End Try
            Next
        Try
            VBMacroUtilities.Print( _
                "    ==Configuration properties==" )
            For Each prop In pitem. _
                ConfigurationManager.Item(1).Properties
                If Not prop.Value Is Nothing Then
                    VBMacroUtilities.Print("    " & prop.Name & _
                        ": " & prop.Value.ToString())
                Else

```

```

        VBMacroUtilities.Print("    " & prop.Name & _
            ": <None>")
    End If
Next
Catch
End Try
Next
End If
End Sub

```

If you look closely, you will see that the property to exclude a file from build in a C++ project is a member of the Configuration object for a ProjectItem object. But for a VB.NET or C# project, the property for excluding a file from build is buried inside the BuildAction property; to exclude the file you set BuildAction to 0, meaning None; to include it, you set it to 1, meaning Compile. But BuildAction is a property of the ProjectItem, not a property of the Configuration object, so there's a slight inconsistency here. But that's okay; the project types are different and your code would have to be different anyway, if you want to set these properties.

Here's a macro that will include or exclude a predefined set of files from a VB.NET project:

```

Private Sub IncExcVBSet(ByVal value As Integer)
    Dim projs As System.Array
    Dim proj As Project
    Dim pitem As ProjectItem
    Dim prop As EnvDTE.Property
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    projs = DTE.ActiveSolutionProjects()
    If projs.Length > 0 Then
        proj = projs.GetValue(0)

        ' Clear out Class1.vb
        pitem = proj.ProjectItems.Item("Class1.vb")
        prop = pitem.Properties.Item("BuildAction")
        prop.Value = value

        ' Clear out Class2.vb
        pitem = proj.ProjectItems.Item("Class2.vb")
        prop = pitem.Properties.Item("BuildAction")
        prop.Value = value

        ' Clear out Class3.vb
        pitem = proj.ProjectItems.Item("Class3.vb")
        prop = pitem.Properties.Item("BuildAction")
        prop.Value = value
    End If
End Sub

```

```

Sub IncludeVBSet()
    IncExcVBSet(1)
End Sub

Sub ExcludeVBSet()
    IncExcVBSet(0)
End Sub

```

Since the code for including and excluding is almost the same, I broke it out into its own private function called `IncExcVBSet`, then I simply called that function from each of the two macros, `IncludeVBSet` and `ExcludeVBSet`. (Notice in the code that you set the `BuildAction` property value to 1 to include the file, and 0 to exclude it.) To try out these macros, add these three files to a VB.NET project, called `Class1.vb`, `Class2.vb`, and `Class3.vb`. Select the project and then run the `ExcludeVBSet` macro. When the files are excluded, you will note two changes: First, when the file is in the editor, the IDE does no syntax checking on the file as you type, and you don't see the drop-down listboxes above the editor showing information about the file. Second, the icon beside the file in the Solution Explorer no longer has an arrow pointing down, meaning the file does not get compiled.



Setting the `BuildAction` property value to 0 to exclude the file from a build is not the same as right-clicking a file in the Solution Explorer and choosing Exclude from Project. The Exclude from Project menu item completely removes the file from the project. Instead, setting the value to 0 is the same as right-clicking on the file, choosing Properties in the popup menu, and setting the Build Action setting to None in the Properties dialog.

When you reinclude the files by running `ExcludeVBSet`, the syntax checking and drop-down listboxes will return, along with the small arrow pointing down in the file icons.

Now here's a similar macro that operates on C++ projects:

```

Private Sub IncExcCPPSet(ByVal exclude As Boolean)
    Dim projs As System.Array
    Dim proj As Project
    Dim pitem As ProjectItem
    Dim prop As EnvDTE.Property
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    projs = DTE.ActiveSolutionProjects()
    If projs.Length > 0 Then
        proj = projs.GetValue(0)

        Dim cfg As Configuration
        pitem = proj.ProjectItems.Item("File1.cpp")
        For Each cfg In pitem.ConfigurationManager
            prop = cfg.Properties.Item("ExcludedFromBuild")

```

```

        prop.Value = exclude
    Next

    pitem = proj.ProjectItems.Item("File2.cpp")
    For Each cfg In pitem.ConfigurationManager
        prop = cfg.Properties.Item("ExcludedFromBuild")
        prop.Value = exclude
    Next

    pitem = proj.ProjectItems.Item("File3.cpp")
    For Each cfg In pitem.ConfigurationManager
        prop = cfg.Properties.Item("ExcludedFromBuild")
        prop.Value = exclude
    Next
End If
End Sub

Sub IncludeCPPSet()
    IncExcCPPSet(False)
End Sub

Sub ExcludeCPPSet()
    IncExcCPPSet(True)
End Sub

```

Manipulating Language-Specific Information

If you read Chapter 5, and if you've been reading this chapter straight through, at this point you might be wondering if you even need language-specific project objects since you can access all the configuration information directly through the Properties collections. The answer is yes!

To explain why, I'll start with the `VSPProject` object, which describes VB.NET and C# projects and is part of the `VSLangProj` namespace. Remember, the properties that I described in the previous sections apply to how a project builds, and they are the items you see in the Property Pages dialog box. The `VSPProject` object, in contrast, gives you information about the project itself, including:

- List of event handlers
- List of imports
- List of references

This information is not available in the property pages. You can access this information only *after* you have a reference to the `VSPProject` object, which you obtain from the `Project` object's `Object` property. Once you have the `VSPProject` object, you can, for example, look at the list of references or even modify the list of references.

Say that you realize you're always using a certain set of assemblies in your programs; in that case, you can write a macro that automatically adds references to all these assemblies, so that you don't have to add them manually. For instance, whenever I develop an add-in, I typically need to add references to the System.Drawing.dll and System.Windows.Forms.dll assemblies. Here's a macro that adds these references for me:

```
Sub AddUsualReferences()  
    Dim projs As System.Array  
    Dim proj As Project  
    Dim vsproj As VSLangProj.VSProject  
    VBMacroUtilities.Setup(DTE)  
    VBMacroUtilities.Clear()  
    projs = DTE.ActiveSolutionProjects()  
    If projs.Length > 0 Then  
        proj = projs.GetValue(0)  
        If proj.Kind <> VSLangProj.PrjKind.prjKindCSharpProject And _  
            proj.Kind <> VSLangProj.PrjKind.prjKindVBProject Then  
            Exit Sub  
        End If  
        vsproj = proj.Object  
        vsproj.References.Add("System.Drawing.dll")  
        vsproj.References.Add("System.Windows.Forms.dll")  
    End If  
End Sub
```

Normally, when you call `References.Add`, you need to pass a full path to the assembly's dll file. However, since the assemblies that I'm adding are part of the global cache, the IDE is able to easily find the assembly in its latest and greatest version.

Moving Forward

In this chapter I expanded on the project and solution information that I introduced in "The Macro and Add-in Models" section of Chapter 5. I also showed you how to:

- Determine the currently active project.
- Add, remove, and modify the items in a project.
- Modify a project's settings.
- Control configurations.
- Work with language-specific information.

All of this information applies to both add-ins and macros. In the next chapter I explain how to manage the Document objects and the various objects that deal with the user interface. These techniques, too, apply to both add-ins and macros.

Programming the Document and User Interface Objects

I begin this chapter by showing you how you can use the `Document` and related objects to create documents and modify, close, and save the documents the IDE user has open. Then I move on to describe an interesting set of objects that let you traverse a hierarchy within a tool window. For example, the Solution Explorer lists all the projects and project items in a hierarchical manner. I show you how you can move about this hierarchy, accessing and modifying the information.

Managing Documents with the Document Classes

The IDE uses different editors for different types of documents, such as a code editor for source files, a visual editor for forms, a data editor for `.resx` resource files, and so on. When an editor is open and contains a document, the IDE maintains a `Document` object. This `Document` object is generic in that its methods and properties apply to all types of documents.



In Chapter 11, I show you how to modify code documents while taking the language into consideration, as well as how to find out a particular document's language (C#, C++, VB.NET). In this chapter, I keep the discussion generic.

Here's a simple macro that lists all the documents:

```
Sub ListDocuments()  
    VBMacroUtilities.Setup(DTE)  
    VBMacroUtilities.Clear()  
  
    Dim d As Document  
    For Each d In DTE.Documents  
        VBMacroUtilities.Print(d.FullName & " " & d.Kind)  
    Next  
End Sub
```

When you run this macro, you will see the names of all the documents that are currently open, along with the document types, which, like so many other objects in the Visual Studio .NET IDE, are GUIDs.

The `Document` object provides basic member functions you would expect a generic document to perform:

Activate. This function will bring the window containing the document to the front, making it the active document. After this, the `DTE.ActiveDocument` object will point to your `Document` object.

Close. This function causes the document to close. You pass `vsSaveChangesYes` to it if you want to automatically save any changes before closing; `vsSaveChangesNo` if you want to abandon any changes (which, believe it or not, is the default if you do not pass any parameters to `Close`); or `vsSaveChangesPrompt` if you would like to prompt the user whether to save any changes for closing.

NewWindow. This function opens a second window for the document; the new window will be added to the `Document.Windows` collection. If you are running in the default tabbed environment (which you set through `Tools` → `Options`, then `Environment` → `General`), you will get a second tab for the single document. The first tab will have a “:1” added to the filename caption; the second tab will have a “:2” added to the filename caption. `NewWindow` is useful primarily for text documents such as C++ source code files. If you attempt to call `NewWindow` on a .vb file, you will get an exception with the message “Not Implemented.”

Undo and Redo. These two functions are equivalent to the IDE user choosing `Edit` → `Undo` and `Edit` → `Redo` in the main IDE menu.

Save. This function saves the document.

Additionally, the `Document` object includes a function that you might not expect to find: `SaveAll`. `SaveAll` saves all open documents that have changed. It takes no parameters.

The collection of `Document` objects is housed within the `DTE.Documents` object, which is an instance of class `Documents`. The `Documents` object has two methods that operate on all documents: `CloseAll` and `SaveAll`. `SaveAll` performs the same as `SaveAll` in the `Document` object, while `CloseAll` takes the same parameters as `Document.Close`. Additionally, the `Documents` object has an `Item` method that lets you find a document based on the document’s name.

Here is a simple macro that finds a particular document and activates it. To try out this macro, you will want to have several files open, including one called `Form1.vb`. Make sure the `Form1.vb` document is not the one in front.

```

Sub ActivateForm1()
    Dim doc As Document
    Try
        doc = DTE.Documents.Item("Form1.vb")
        doc.Activate()
    Catch
    End Try
End Sub

```

Interestingly, you can instead pass the document's full path and filename to the `Item` function, as shown in the following line of code:

```
doc = DTE.Documents.Item("C:\dev\Projects\VBFormTest1\Form1.vb")
```

Therefore, depending on your situation, you can use either just the filename or the full path and filename.

The Form Editor and Documents

If you have a form open and you run the earlier `ListDocuments` macro, you will see two documents for the single form. One is the form editor, whose name will have a `.resx` extension; the other is the document for the code, and will have either a `.vb` or `.cs` extension, depending on whether the form is a Visual Basic or C# form.

You can find out the active document (that is, the document that has the focus) by checking the `DTE.ActiveDocument` property. When you're dealing with a visual form, you're going to see something a little strange. Take a look at this macro:

```

Sub ShowActiveDocument()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    VBMacroUtilities.Print(DTE.ActiveDocument.Name)
End Sub

```

If you open up both a form editor and a code editor for a single form (such as the default `Form1.vb` in a VB.NET Windows application) and switch first to the code editor and run the preceding macro, you will see `Form1.vb` (the code document) as the active document. Then if you switch to the form, instead of seeing `Form1.resx`, you will again see the `Form1.vb` as the active document.

Now suppose you use the `DTE.Documents.Item` method to locate both the form editor document and the code editor document, such as `Form1.resx` and `Form1.vb`, respectively. If you call the `Form1.vb Document` object's `Activate` method, you will actually activate the form designer, not the code editor window, as you might expect. Furthermore, if you call the `Form1.resx Document` object's `Activate` method, you will get an exception. So be careful when working with the `Document` and `Documents` objects while dealing with forms; also use care if you expect this odd behavior, which has been documented by Microsoft and is expected to change it in the next edition of Visual Studio .NET.

Opening or Creating a Document

To open an existing document or to create a new document, use the `ItemOperations` object, which is a property of the DTE object. The `ItemOperations` object includes an `OpenFile` member function and a `NewFile` member function.

To use the `OpenFile` function, pass the full path and filename as a string for the first parameter, and, optionally, a constant to denote the type of file you wish to open. Before I show you about these types, look at this quick sample of a call to `OpenFile`:

```
Sub DemoOpenFile()  
    Try  
        DTE.ItemOperations.OpenFile("c:\myfile.txt")  
    Catch  
        MsgBox("Could not open the file.")  
    End Try  
End Sub
```

As you can see, I wrapped the `OpenFile` function with a `Try/Catch` block to ensure that I properly handled the situation where the file cannot be opened, as would be the case if the file didn't exist.

You can also include a second parameter to the call to `OpenFile`. Take a look at this code:

```
DTE.ItemOperations.OpenFile("C:\dev\MyProject\Form1.vb", _  
    EnvDTE.Constants.vsViewKindDesigner)
```

then compare it to this code:

```
DTE.ItemOperations.OpenFile("C:\dev\MyProject\Form1.vb", _  
    EnvDTE.Constants.vsViewKindCode)
```

As you can see, both sets of code open the same file; however, the first specifies `vsViewKindDesigner` for the second parameter, while the second specifies `vsViewKindCode` for the second parameter. The first version of the code causes the IDE to open the Visual Basic code file in designer mode, so that you can draw components on the form. The second version of the code causes the IDE to open the file as a code file, so you can modify the code directly. For the first version to work, however, you must have an actual VB.NET project, including a `.resx` file containing the form information.

If you try to open a file that is not a form file (such as any C++ file) using the `vsViewKindDesigner` parameter, the IDE will give an error, allowing you to look at the XML source for the file. Of course, the file doesn't have any XML; it's a C++ source file, and so the error message you see will be that the XML is not well formed.

You can also use the `ItemOperations` object to create a new file by calling the `NewFile` function. Here's a sample:

```
DTE.ItemOperations.NewFile("General\HTML Page", "myfile.html")
```

The first parameter corresponds to the items you can find in the New File dialog box. From the main IDE, if you choose File-New⇨File, the New File dialog box will open, showing a list of categories on the left side. When you click on one of the categories, you will see a list of file templates on the right side. The first parameter to `NewFile` is a string of the form "Category\Template"; you replace Category with the name of a category you can find on the left side of the New File dialog box, and you replace Template with the name of the template on the right side of the New File dialog box. (Of course, when you run the `NewFile` function, the New File dialog box doesn't actually appear.)

The second parameter to `NewFile` is simply a suggestion for a new filename. By that I mean, when the IDE user chooses File⇨Save, the Save File As dialog box will open, and the filename you specify in the second parameter will appear as the default filename.



When you call `NewFile`, the file will not be written to disk until the IDE user saves the file, or you save the file through a macro using the `Document.Save` function. Further, the IDE ignores the path name when you specify a filename for the second parameter in the call to `NewFile`. When the IDE user chooses to save the file, the File Save As dialog box will open in the current working directory, and the default filename will be that given in the second parameter to `NewFile`. Consequently, you can actually call `NewFile` twice with the same filename. Just be careful when you close a file that you called with `NewFile` because if the IDE user makes no changes to the file and then closes it, the file will *not* be saved to disk; that is, *no file will be created!*

You can also call `NewFile` with only a template name and no filename to create an unnamed file:

```
DTE.ItemOperations.NewFile("General\HTML Page")
```

In this case, the IDE user would choose the filename when he or she saves the file, or you would in your macro when you save the file.

In order for your macro to save a file that the macro creates, the `NewFile` function returns an instance of `Window`, from which you can access the `Document` object. The `Document` object has a `Save` method. Remember, since the second parameter to `NewFile` is simply a suggestion to the IDE user, if you plan to save the file from your macro, you do not need to specify a second parameter to `NewFile`, since the name will be ignored anyway. (If you do specify a name for the second parameter, it will not be used anywhere.) Next shown is an example macro that creates a new file and saves it. You can see that I did not provide a name for the second parameter to `NewFile`:

```
Sub DemoNewFile()
    Dim win As Window
    win = DTE.ItemOperations.NewFile("General\HTML Page")
    win.Document.Save("c:\myfile2.html")
End Sub
```

You may notice that if you open the New File dialog box, several items will not be present. For instance, the dialog box contains no Visual Basic items, such as forms and modules. When interacting with the IDE, you right-click on a project, and in the popup menu choose Add→Add New Item. The Add New Item menu item corresponds to a separate function in the `ItemOperations` object called `AddNewItem`. You call this item just as you would `NewFile`, passing a string representing the item in the Add New Item dialog box. Since the tree on the left has a hierarchy, you separate the items in the hierarchy with a backslash. Thus, if you want to add a new module to a VB.NET project, you would make a call like this:

```
DTE.ItemOperations.AddNewItem("Local Project Items\Code\Module")
```

This creates a file with a default filename and immediately adds the file to the project. Unlike `NewFile`, `AddNewItem` automatically saves the file to disk. In this code, because I did not specify a filename, the IDE gives the filename a default name such as `Module1.vb`. You can also specify a filename in the second parameter:

```
DTE.ItemOperations.AddNewItem ("Local Project Items\Code\Module", _  
    "newmodule.vb")
```

In this case, the filename is not a suggestion; the IDE actually saves the file into the project directory with the name you specify.

Unlike the `NewFile` function, which returns a `Window` object, the `AddNewItem` function returns a `ProjectItem` object.

Processing Text

The `Document` object gives you access to general information about a document, but contains no text-specific information and operations. By keeping the `Document` object generic, you can use it with any type of document, not just text documents such as source files. To work with the text in a text document, you have at your disposal several other classes.

Part of the fun of working with text is keeping all the classes straight. To that end, I first provide you with a list of classes you can use, along with a brief description of each; then I give you the details on how to work with them.

TextDocument. This is the primary entry point into the other objects. Through a `Document` object you can obtain a `TextDocument` object, which includes a few general methods and properties. However, most of the text-processing actions you will perform are available through the objects in the rest of the list; you obtain these objects through this `TextDocument` object.

TextSelection. This object includes all the operations you would expect to perform on text, focusing primarily on selections. You can manipulate the current *selection point* of the text, that is, where the blinking cursor is in the text on the screen and which text is highlighted; you can format the text, and so on. (You can also access the `TextSelection` object directly from the `Document` object through the `Selection` property.)

EditPoint. An `EditPoint` object represents a particular position within a block of text. It gives you direct access to the text in a document without being affected by the text as it appears in the editor window due to the IDE user's editor settings. Further, with an `EditPoint` object, you can modify the text directly without affecting the insertion point.

TextRange. A `TextRange` object holds a pair of `EditPoint` objects, representing a range of text.

TextRanges. This is a collection of `TextRange` objects.

VirtualPoint. This is the whitespace in a code editor where there is no text.

Now look at this short example of the `TextDocument` and `TextSelection` objects. You can see that I obtained the `TextDocument` object through the `Object` property of the `Document` object; and from the `TextDocument` object I obtained the `TextSelection` object through the `Selection` property.

```
Sub GetText ()
    VMacroUtilities.Setup (DTE)
    VMacroUtilities.Clear ()
    Dim doc As Document
    doc = DTE.ActiveDocument
    Dim textdoc As TextDocument
    textdoc = doc.Object
    textdoc.Selection.SelectAll ()
    VMacroUtilities.Print (textdoc.Selection.Text)
End Sub
```

This macro simply grabs the active document and copies the text to the output window. I didn't put any error checking in it, so make sure your active document is a text document such as a source file. You can see that I obtained the `TextDocument` object by accessing the `Document` object's `Object` property. Also note that the `TextDocument` object has a property, called `Selection`, which in turn has a method called `SelectAll`. This method selects all the text in the document. The selection of the text is not a behind-the-scenes operation; you will actually see the text in the document become highlighted. Finally, I printed out the selection by accessing the `Selection` property's `Text` property (the `Text` property is a string).

The `GetText` macro simply obtains the text in the document; it does not modify the text, whereas the `TextDocument` also lets you modify the text. The `TextDocument` class includes a `ReplacePattern` method, which lets you do global replacements in the document using regular expressions. (For an introduction to regular expressions, see "Finding an Item Using Regular Expressions" later in this chapter.) Here's an example:

```
Sub ReplaceText ()
    Dim doc As Document
    doc = DTE.ActiveDocument
    Dim textdoc As TextDocument
    Dim trange As TextRanges
    textdoc = doc.Object
```



```

textdoc.ReplacePattern("/(.)/", _
    "zzz", _
    EnvDTE.vsFindOptions.vsFindOptionsRegularExpression, trange)
End Sub

```

This replaces any instance of a slash, followed by a single character, followed by a slash, with the string “zzz”.

The `TextSelection` object lets you manipulate text using many of the features for which you would normally use an editor. For example, you can move the insertion point around, as in this code snippet:

```

Dim td As TextDocument = ActiveDocument.Object
td.Selection.WordRight()

```

This code moves the insertion point one word to the left:

```

Dim td As TextDocument = ActiveDocument.Object
td.Selection.WordLeft()

```

If you optionally pass `True` for the first parameter to `WordRight` or `WordLeft`, instead of simply moving the insertion point right or left, you will also select the text. Thus, this selects three words to the right and then selects one word to the left, which has the net effect of unselecting the rightmost word, resulting in two words being selected:

```

Sub DemoSelectSeveral()
    Dim td As TextDocument = ActiveDocument.Object
    td.Selection.WordRight(True)
    td.Selection.WordRight(True)
    td.Selection.WordRight(True)
    td.Selection.WordLeft(True)
End Sub

```

You can also specify a number for the second parameter to represent the number of words to move or select. Thus, you could write the preceding `DemoSelectSeveral` macro like so to have the same effect:

```

Sub DemoSelectSeveral2()
    Dim td As TextDocument = ActiveDocument.Object
    td.Selection.WordRight(True, 3)
    td.Selection.WordLeft(True)
End Sub

```

You can also move the insertion point up or down a line by calling `Selection.LineUp` or `Selection.LineDown`, optionally passing `True` for the first parameter if you want to select the text in the process, and optionally passing a count for the second parameter, just as you can with `WordRight` and `WordLeft`.

In the context of the `TextSelection` object, the term *collapse* means to remove a selection point, leaving the insertion point in place. Thus, if you click in a document on the letter “a” in the word “apple,” then hold down the Shift key and press the right-arrow key twice (to select two characters), and then collapse the selection, the insertion point will end up between the two p’s in “apple.” You can collapse a selection programmatically, by calling `Collapse`, like so:

```
Sub DemoCollapse()  
    Dim td As TextDocument = ActiveDocument.Object  
    td.Selection.Collapse()  
End Sub
```

You can also move to the start or end of the current line using the `Selection.StartOfLine` and `Selection.EndOfLine` functions. When you’re actually typing in the code editor, you have probably noticed that when you press the Home key, initially, the insertion point moves to the first nonwhitespace character; that is, if the code is indented, the insertion point moves to the start of the code, not the left-most column. When you press Home a second time, the insertion point moves to the first column. The `StartOfLine` function provides for this same functionality. The first parameter, which is optional, can be either `vsStartOfLineOptionsFirstColumn`, which moves the insertion point to the first column, or `vsStartOfLineOptionsFirstText`, which moves the insertion point to the first nonwhitespace column, where the text begins. (Both of these constants are part of the `vsStartOfLine` enumeration.) You can also optionally pass `True` as a second parameter to `StartOfLine` if you want to select the text between the current insertion point and the start of the line.

Here’s a sample piece of code that moves to the first column, selecting the text in the process:

```
Dim td As TextDocument = ActiveDocument.Object  
td.Selection.StartOfLine( _  
    vsStartOfLineOptions.vsStartOfLineOptionsFirstColumn, True)
```

And here’s a sample piece of code that moves to the first nonwhitespace column, not selecting the text:

```
Dim td As TextDocument = ActiveDocument.Object  
td.Selection.StartOfLine( _  
    vsStartOfLineOptions.vsStartOfLineOptionsFirstText)
```

If you pass no parameters to `StartOfLine`, the default is `vsStartOfLineOptionsFirstColumn`; that is, the insertion point will move to the first column.

The `EndOfLine` function does not have a parameter analogous to the first parameter in `StartOfLine`. Instead, the only parameter to `EndOfLine` is optional, and it’s a Boolean representing whether to select the text. Here’s a sample line that selects from the current position to the end of the line:

```
Dim td As TextDocument = ActiveDocument.Object  
td.Selection.EndOfLine()
```

The `TextSelection` object also lets you modify the text. The following macro will insert text at the current position in the document:

```
Sub DemoInsertText()
    Dim td As TextDocument = ActiveDocument.Object
    td.Selection.Insert("Hello")
End Sub
```

If text is selected, you can have complete control of how you want to insert the new text by passing an optional second parameter to `Selection.Insert`. Here are your choices:

- vsInsertFlagsCollapseToEnd.** This will replace the selected text with the new text and then move the insertion point to the *end* of the new text, leaving the new text unselected.
- vsInsertFlagsCollapseToStart.** This will replace the selected text with the new text and then move the insertion point to the *beginning* of the new text, leaving the new text unselected.
- vsInsertFlagsContainNewText.** This will replace the selected text with the new text and then leave the new text selected, without changing the position of the insertion point. Thus, if you highlight some text from right to left, causing the insertion point to be at the beginning of the text, then after calling `Insert`, specifying `vsInsertFlagsContainNewText` for the second parameter, after the insert the insertion point will remain at the beginning of the new text.
- vsInsertFlagsInsertAtEnd.** This will add the new text to the end of the selection, keeping the original text present and selected, adding the new text to the selection. Thus, if your selected text is "HELLO " and you call `Insert("There", vsInsertFlagsInsertAtEnd)`, then the new text will be "HELLO THERE", with the entire "HELLO THERE" highlighted.
- vsInsertFlagsInsertAtStart.** This has the same effect as `vsInsertFlagsInsertAtEnd`, except the text you are inserting will be inserted at the beginning of the selection.

The default, without specifying a second parameter, is `vsInsertFlagsCollapseToEnd`. Here's a sample showing `vsInsertFlagsContainNewText`:

```
Sub DemoInsertTextContainNewText()
    Dim td As TextDocument = ActiveDocument.Object
    td.Selection.Insert("Hello", _
        vsInsertFlags.vsInsertFlagsContainNewText)
End Sub
```

Deleting is a bit less sophisticated than inserting. If you call `Document.Delete` with no parameters, you will delete the current selection. Or, if there is no current selection, you will delete the current character that's to the right of the insertion point. Here's a macro showing `Delete`:

```
Sub DemoDelete()
    Dim td As TextDocument = ActiveDocument.Object
```

```
td.Selection.Delete()
End Sub
```

You can optionally pass a count to `Delete`, specifying how many times to delete:

```
td.Selection.Delete(3)
```

If text is currently selected, then that text will be deleted first, representing the first iteration. Then for each remaining iteration, the characters to the right will be deleted. Thus, if you have the text “HELLO” and you have HE selected, calling `Delete(3)` will first delete the selected text, HE, then the next character, L, and then the next character, another L, leaving only “O.”

To delete the current word, you first have to select it and then delete it. To select the current word, first move to the left of the word without selecting, then move to the right while selecting. Here’s a macro that deletes the current word:

```
Sub DeleteWord()
    Dim td As TextDocument = ActiveDocument.Object
    td.Selection.WordLeft()
    td.Selection.WordRight(True)
    td.Selection.Delete()
End Sub
```

Deleting a line is slightly easier, because the `TextSelection` object has a `SelectLine` function:

```
Dim td As TextDocument = ActiveDocument.Object
td.Selection.SelectLine()
td.Selection.Delete()
```

You can also change the text itself. This code will capitalize the first letter of every word in the selection text, while making the other words lowercase:

```
Dim td As TextDocument = ActiveDocument.Object
td.Selection.ChangeCase(vsCaseOptions.vsCaseOptionsCapitalize)
```

Thus, if you highlight the following comment in your code:

```
When run, the Add-in wizard prepared the registry for the Add-in.
```

and then run the preceding code, the comment will change into:

```
When Run, The Add-In Wizard Prepared The Registry For The Add-In.
```

This code will change all letters into capitals; thus, if you select the word “Prepared,” the word will turn into “PREPARED.”

```
Dim td As TextDocument = ActiveDocument.Object
td.Selection.ChangeCase(vsCaseOptions.vsCaseOptionsUppercase)
```

And this will make all words in the selected text lowercase:

```
Dim td As TextDocument = ActiveDocument.Object
td.Selection.ChangeCase(vsCaseOptions.vsCaseOptionsLowercase)
```

As you can imagine, the `ChangeCase` member function is best used in a comment, since, typically, you won't want your macro or add-in changing the case of code, especially if the code is C++.

You can also perform standard copy, cut, and paste actions. The functions are `Copy`, `Cut`, and `Paste`, each with no parameters. Using some of the functions I described previously in this section, you can copy a line of text and paste a copy just below the line, like so:

```
Sub CopyLine()
    Dim td As TextDocument = ActiveDocument.Object
    td.Selection.SelectLine()
    td.Selection.Copy()
    td.Selection.Collapse()
    td.Selection.Paste()
End Sub
```

This `CopyLine` macro might look a little strange to you because I didn't move the cursor down a line. The reason is that the `SelectLine` selects the entire line, putting the insertion point at the start of the next line. So after I call `Copy`, I collapse the selection, which leaves the insertion point at the start of the line below where I started. Thus, the code implicitly moves the insertion point down a line. Then I just call `Paste`.

In addition to all the functions I just described, the `TextSelection` has numerous other functions that let you edit the text in a document. I encourage you to take a look at the online help to see the whole array of possibilities. To find the online help entry, open the Index tool window for the online help, type Document Object, and press Enter. In the Index Results pane, double-click Document Object Properties, Methods, and Events. This will open the online help entry showing you all the members available to you in the Document object.

The EditPoint, TextPoint, and VirtualPoint Objects

An `EditPoint` is an object that serves two purposes: First, it represents a single position within a document. Second, it allows you to modify the text in a document without affecting the insertion point and selection of the document.

To see this second purpose, try opening a text file (such as a source code file). Make sure the top of the document is in view, select some text in the middle of the document, then run this macro:

```
Sub TestEditPoint()
    Dim textdoc As TextDocument
    textdoc = DTE.ActiveDocument.Object
    Dim ep As EditPoint
```

```

    ep = textdoc.CreateEditPoint
    ep.Insert("Hello")
End Sub

```

When you run it, you will see your selected text remains selected and unchanged, while the word “Hello” gets inserted at the very beginning of the document.

In this next macro, I created an `EditPoint` object based on the starting position of the current document’s `TextDocument` object. You can also create an `EditPoint` object based on the currently selected text. Go ahead and select some text in your document and then try running this macro:

```

Sub TestEditPoint2()
    Dim ep As EditPoint
    ep = DTE.ActiveDocument.Selection.ActivePoint.CreateEditPoint
    ep.Insert("Hello")
End Sub

```

This code inserts text at the current insertion point, again without disturbing the selection.

The `EnvDTE` namespace also includes two classes related to `EditPoint`, called `TextPoint` and `VirtualPoint`. Think of `EditPoint` as an editable position in the document, whereas `TextPoint` and `VirtualPoint` objects show a position, but you cannot edit the text. If you have a `TextPoint` or a `VirtualPoint` representing a position in the document, and you want to edit the text at that position, you must obtain from the `TextPoint` or `VirtualPoint` object an `EditPoint` object. You do so by calling either `TextPoint.CreateEditPoint` or `VirtualPoint.CreateEditPoint`.

Here are some different positions in a document, the way to find the position, and the type of object associated with the position. For each of these, you can call the returned object’s `CreateEditPoint` if you want an editable version of the position.

Start of document. Use `TextDocument.StartPoint`, which returns a `TextPoint` object.

End of document. Use `TextDocument.EndPoint`, which returns a `TextPoint` object.

Start of selected text. Use `Document.Selection.TopPoint`, which returns a `VirtualPoint` object.

End of selected text. Use `Document.Selection.BottomPoint`, which returns a `VirtualPoint` object.

Anchor Point. This refers to the position in the text at which the insertion point started when the IDE user selected the text. It will correspond to either the start or the end of the selected text, depending on which direction the IDE user highlighted the text. Use `Document.Selection.AnchorPoint`, which returns a `VirtualPoint` object.

Active Point. This refers to the position at which the insertion point ended when the IDE user selected the text. Use `Document.Selection.ActivePoint`, which returns a `VirtualPoint` object.

First displayed character in the pane. Since the text is displayed in a pane, use `TextPane.StartPoint` to determine the first visible character in the pane. (See “Working with Multiple Windows and Panes” later in this chapter for more information on panes.)

You don’t always have to call `CreateEditPoint` when working with the positions in the document. If you’re not interested in modifying text at a particular point, you can simply use that point’s `TextPoint` or `VirtualPoint` object. Look at this macro:

```
Sub VBCommentSelection()
    Dim start As EditPoint
    Dim theend As VirtualPoint
    start = DTE.ActiveDocument.Selection.TopPoint.CreateEditPoint()
    theend = DTE.ActiveDocument.Selection.BottomPoint
    start.StartOfLine()
    Do While (start.LessThan(theend))
        start.Insert(" ")
        start.LineDown()
        start.StartOfLine()
    Loop
End Sub
```

This macro puts Visual Basic comment marks (an apostrophe) at the start of each line that contains selected text. The macro uses two positions, the start point and the end point of the selected text. It then moves to the beginning of the starting line and inserts a comment mark, then moves down and back to the start of the line.

HANDLING UNDO CONTEXTS

If your macro or add-in makes multiple changes to a document and you want to enable the user to undo all the changes with a single press of Ctrl+Z or a single selection of Edit→Undo, you can wrap the changes in an *undo context*. First, you call `DTE.UndoContext.Open`, passing a string representing the overall operation. Here’s a modified form of an earlier macro, `ReplaceText`, with an undo context. After you run this macro, you can undo its results simply by pressing Ctrl+Z one time. (With the previous version of the macro, you had to press Ctrl+Z once for each replacement.)

```
Sub ReplaceText2()
    Dim doc As Document
    doc = DTE.ActiveDocument
    Dim textdoc As TextDocument
    Dim trange As TextRanges
    DTE.UndoContext.Open("Undo Replacement")
    textdoc = doc.Object
    textdoc.ReplacePattern("/(.)/", _
        "zzz", _
        EnvDTE.vsFindOptions.vsFindOptionsRegularExpression, trange)
    DTE.UndoContext.Close()
End Sub
```



After calls to `LineDown` and `StartOfLine`, the `EditPoint` changed, but the original highlighted text in the document did *not*. By calling `CreateEditPoint`, you are effectively creating an object that is disconnected from the original selection.

As the macro proceeds, it compares the start `EditPoint` object to the end `EditPoint` object, using the `LessThan` method. This method compares the positions within the document, and if the object is earlier in the document than the parameter (that is, if the start point is less than the end point), the function returns `True`. (You also have at your disposal `GreaterThan` and `EqualTo` functions, which are members of `EditPoint`, `TextPoint`, and `VirtualPoint`.)



There is really little difference between a `TextPoint` and a `VirtualPoint`, except that a `VirtualPoint` can appear at the very end of a line, representing the whitespace to the right of a line.

Working with Multiple Windows and Panes

In Visual Studio .NET, the IDE user can have multiple windows open, each containing a copy of the same source code file. Additionally, the user can split each window vertically into two panes, allowing two separate views of the same file. Changes to a document affect not only the current pane or window, but all panes and windows containing the same document. Such changes include undo and redo actions. However, each pane and window maintains its own insertion point and highlighted text. You can highlight text in one pane, while highlighting separate text in another pane or window for the same document. Similarly, you can have the insertion point at one place in one pane or window and in another place within the same document but in another pane or window.



When you have multiple windows showing the same document, the `Document.Selection` object corresponds to the window that is active. Further, if the windows have multiple panes, the `Document.Selection` object will correspond to the pane that is active.

The following macro creates a new window for the currently active document. (Note, however, that the `NewWindow` function is not available for VB.NET code files; thus, this macro will function only for C++ and C# code files. Running it for a VB.NET code file will generate a “Not implemented” error message.) In this macro I included a `Try/Catch` block, because you cannot call `NewWindow` for every type of window.

```
Sub CreateNewWindow()
    Dim doc As Document
    Try
        doc = DTE.ActiveDocument
        doc.NewWindow()
    Catch e As System.Exception
```



```
        MsgBox(e.Message)
    End Try
End Sub
```

The Document object includes a Windows property that contains a collection of Window objects. Normally this collection consists of only one Window object, representing the single window containing the document. But if you create a second window using the preceding macro, then you will find two Window objects in the Windows collection.

When you create a second window for a single document, you will see either two tabs, if you're running in tabbed mode, or two windows, if you're running in MDI mode. (The default is tabbed mode.) However, even though you have two Window objects, you still have only a single Document object. You can see this by running the following macro:

```
Sub ListDocumentsWithCount()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()

    Dim d As Document
    For Each d In DTE.Documents
        VBMacroUtilities.Print(d.FullName & " " & d.Windows.Count)
    Next
End Sub
```

If you run the previous macro, `CreateNewWindow`, and then run this macro, `ListDocumentsWithCount`, you will see that you still have only a single Document object for the two windows. The `ListDocumentsWithCount` macro prints out the name of each document, along with the `Windows.Count` property, which tells how many Window objects are in the Windows collection.

You can also split a single window into two panes. To do this, choose `Window→Split` (and then `Window→Remove Split` to remerge the window).

To find out the panes, start with a Window object. From there, you obtain a `TextWindow` object, which includes a `Panes` collection. The following macro shows you how to do this:

```
Sub ListPanes()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim tw As TextWindow
    Dim panes As TextPanes
    Dim pane As TextPane

    tw = ActiveDocument.Windows.Item(1).Object
    panes = tw.Panes
    For Each pane In panes
        VBMacroUtilities.Print(pane.Height)
    Next
End Sub
```

This macro operates on the first window of the active document. Make sure your active document is a text window (such as a source code file); otherwise, you'll get an exception. You also might try splitting the window into two panes to see the results.

The macro also prints out the height of each pane. The `PANE` object has both a width and a height, and, interestingly, the values of the width and height are in terms of characters, not pixels. Thus, the width is the number of columns the editor window's pane can show, and the height is the number of rows it can show.

Navigating the User Interface Hierarchy

Many of the window objects in the IDE contain a hierarchy of some sort. For example, the Solution Explorer contains a hierarchy that lists, first, the projects, and under each project the project items. Inside the `EnvDTE` namespace is a class called `UIHierarchy`, which contains the elements in a hierarchy. To obtain the `UIHierarchy` instance for a particular window (such as for the Solution Explorer window), first access the `Window` object for the window, and from the `Window` object access the `Object` property. The `Object` property in such windows is an instance of `UIHierarchy`.

If the window is not one that shows a sort of hierarchy (for example, the Output window does not), then the `Object` property is *not* an instance of `UIHierarchy`. Therefore, before you can use the `Object` property, you must check its type. When I was doing research for this chapter, I first checked the GUID of the `Object` property, expecting it to be the same each time a `UIHierarchy` was present. That would have made sense, since the `UIHierarchy` is a class. But I was wrong. The `Object` property does, in fact, refer to a different type of COM object for each window, thus the Solution Explorer window and the Server Explorer window, both of which have a `UIHierarchy` available, have different GUIDs for their `Object` properties. So instead of checking the GUID, the correct way is to call the VB.NET function `TypeName`.

Interestingly, even though `TypeName` returns the correct string (such as "`UIHierarchy`"), you cannot call `typeof`, as `typeof` will return a COM type, but not the name `UIHierarchy`. Therefore, in my code I am using `TypeName`.



Before using the `UIHierarchy` object, make sure the `Object` property for the window is not set to `Nothing`. If the window has no hierarchy (or if the window can support a hierarchy but does not currently have a hierarchy showing), then the `Object` property will be set to `Nothing`, rather than to an instance of `UIHierarchy`.

The following macro gets the active window, then the `Object` property, and makes sure the `Object` property is not null (or, in VB.NET terminology, `Nothing`); next the code gets the type name of the `Object` property. If the type name is `UIHierarchy`, the code traverses through the hierarchy by calling a recursive function. Here's the macro:

```
Sub Traverse(ByVal indent As Integer, ByVal items As UIHierarchyItems)
    Dim subitem As UIHierarchyItem
    For Each subitem In items
        VBMacroUtilities.Print(" ".PadLeft(indent * 4) & _
```

```

        subitem.Name & " " & subitem.IsSelected)
    If Not subitem.UIHierarchyItems Is Nothing Then
        Traverse(indent + 1, subitem.UIHierarchyItems)
    End If
Next
End Sub

Sub UIHierTest()
    Dim window As Window
    VBMacroUtilities.Setup(DTE)
    window = DTE.ActiveWindow
    If Not window.Object Is Nothing Then
        If TypeName(window.Object) = "UIHierarchy" Then
            Dim UIH As UIHierarchy = window.Object
            VBMacroUtilities.Print(window.Caption & " " & _
                window.ObjectKind)
            Traverse(0, UIH.UIHierarchyItems)
        Else
            VBMacroUtilities.Print(" Not a UIHierarchy type: " _
                & TypeName(DTE.ActiveWindow.Object))
        End If
    Else
        VBMacroUtilities.Print(" No hierarchy is available.")
    End If
End Sub

```

This macro is primarily for informational purposes, as it does not modify anything in the IDE. To try out the macro, make sure you have a solution open. Then, open the Solution Explorer window, making sure it's active (that is, its title bar is showing as the active window). Run the macro by double-clicking the macro name in the Macro Explorer.



When you click on a window and then double-click a macro in the Macro Explorer, the `DTE.CurrentWindow` object will still contain a reference to the window you clicked prior to running a macro. This is an advantage, as it allows you to select a window and then run a macro without causing the `DTE` object to treat the Macro Explorer as the active window. If you want to traverse the Macro Explorer's hierarchy, instead of using the `CurrentWindow` property, you can use `DTE.Windows.Item(Constants.vsWindowKindMacroExplorer)`.

The macro will traverse through the hierarchy, starting at the top, listing the names in the tree. Here's a sample output:

```

Solution Explorer - Solution Items {3AE79031-E1BC-11D0-8F78-00A0C9110057}
MyProjects False

```

```

CSharpWinApp1 False
  References False
    System False
    System.Data False
    System.Drawing False
    System.Windows.Forms False
    System.XML False
  App.ico False
  AssemblyInfo.cs False
  Form1.cs True
  test.xml False
  XMLSchema1.xsd False
DatabaseProject1 False
  Change Scripts False
  Queries False
  Database References False
    FOXPRO.C:\dev\Projects\BOOKS.DBC False

```

The first line gives the name of the window (just to make sure I clicked the right one) and then the GUID of the Object (stored in the `ObjectKind` property). I printed out this GUID so you can see that the GUID is different for different windows. Next is the name of the solution, which corresponds to the first item in the tree. This solution has two projects in it, a C# program and a database application for a FoxPro database. I listed the names of each project, along with the items inside the project. Now you'll notice that the hierarchy contains an item for *every* item in the Solution Explorer, including folders.

In addition to the name of each item in the tree, the macro prints out the value of the item's `IsSelected` property. This is where your macro can determine whether the user has clicked on the item. You can see in my list that the C# form file, `Form1.cs`, is the selected item. If you select multiple items (using the Shift or Ctrl keys), all the selected items will have a True by them.

A quick aside on the recursive subroutine `Traverse`: Let me begin by explaining how the `UIHierarchy` object works. It contains a property called `UIHierarchyItems`, which is a list of `UIHierarchyItem` objects. Each `UIHierarchyItem` object, in turn, contains a `UIHierarchyItems` list. That means that the top item in the hierarchy is a `UIHierarchy` object, whereas all the other items in the hierarchy are `UIHierarchyItem` objects. Since I wanted to write my `Traverse` routine to handle the general case, I made its parameter a `UIHierarchyItems` collection. That way, whether I'm dealing with a `UIHierarchy` object or a `UIHierarchyItem` object, I can always pass in the object's `UIHierarchyItems` property.

Also, the `Traverse` routine takes an integer, which represents an indentation level. You can think of this as how deep the current item is in the recursion. But the way I use the item in the macro is to create a string of spaces whose length is four times the indentation level. (I do that with the `PadLeft` function; there are probably other ways to do the same thing.) Thus, when I print out the items, they have an indentation that matches the indentation in the window's tree.



You can traverse a window's hierarchy even if the window is not currently showing. For example, if you close the Solution Explorer window, the IDE is still aware of the Solution Explorer, so you can still traverse its hierarchy.

Finding a Hierarchy Item

If you want to find a particular hierarchy item, you can start by locating the window for the hierarchy. Do so via the usual method, such as this:

```
Dim win As Window = DTE.Windows.Item( _  
    Constants.vsWindowKindSolutionExplorer)
```

The `Constants` class contains a list of window types, all of which start with the `vsWindowKind`. The easiest way to locate the constant name is to know the name of the window (such as Solution Explorer or Server Explorer), then type the word `Constants` into the code editor, next the dot, and then slowly type the first few letters of `vsWindowKind`. As you do, a popup window will appear showing a list of all the members of the `Constants` class; the list will be centered on the names that start with `vsWindowKind`. You can then scroll through the list, looking for the name of the window you're trying to locate. The name will typically be the same as the window's caption, but without the spaces. Thus, the Solution Explorer's constant is `vsWindowKindSolutionExplorer`, and the Server Explorer's is `vsWindowKindServerExplorer`.

After you have the `Window` object, grab the object's `UIHierarchy` object through the `Object` property:

```
Dim uih As UIHierarchy = win.Object
```

After you have the `UIHierarchy` item, you have some choices, depending on your situation. If you know the name of the item (such as `Form1.cs` in the `CSharpWinApp1` project), you can quickly find the item using the `UIHierarchy` object's `GetItem` function. For this function, however, you need the *full path* to the item inside the hierarchy. So in the case of the `Form1.cs` item in the `CSharpWinApp1` project, you would use a couple of lines such as these:

```
Dim item As UIHierarchyItem  
item = uih.GetItem("MySolution\CSharpWinApp1\Form1.cs")
```

You can see that I had to give the full path to the `Form1.cs` item, including the root node in the hierarchy, which is the name of the solution. For all the items except the root, the name is the same as it appears in the window's tree. But the name is not as it appears in the window for the root node; instead, the name is simply that of the solution. Interestingly, if you put an invalid name in the string, you will receive an error message that reads, "The parameter is incorrect."

Finding an Item Using Regular Expressions

If you've never worked with regular expressions, I strongly encourage you to learn about them. Regular expressions make any string or text processing amazingly simple, once you understand the somewhat cumbersome syntax. For years, regular expressions were something that primarily Unix gurus understood; the rest of us simply didn't like to admit that we knew nothing about them.



In this book, I can give you only a bit of introductory material on regular expressions so I recommend you get a copy of *Mastering Regular Expressions*, by Jeffrey E.F. Friedl (O'Reilly & Associates, 2nd ed., July 2002). This is, by far, the best book on the topic, and one that every programmer should own (in addition to the book in your hands, of course).

The beauty of regular expressions is that you can write a complex pattern and then determine if a string matches it. This is a bit like the wildcard patterns you can use in a DOS window, such as *.txt, which refers to every filename ending in ".txt."

The .NET framework includes a set of full-featured regular-expression classes that makes regular expression handling remarkably easy. To use the classes, you first define your search string, passing it into the constructor of the `Regex` class:

```
reg = New Regex(".*\.cs")
```

This line of code defines a pattern that matches any filename ending in the string ".cs". The beginning of the pattern is `.*`, which means *any string of characters*. (It's equivalent to the single asterisk (*) in DOS filenaming; however, in this case, the dot character refers to *any character*, and the asterisk means any number of the previous special character. Thus `.*` means *any number of any character*.) Since the dot character is special, you use a backslash followed by a dot if you really mean a dot. Thus, `\.cs` means the literal string ".cs". And so this particular `Regex` object is specifying a pattern that will match any string ending in ".cs".



The regular expression classes live in the `System.Text.RegularExpressions` namespace. Thus, to access the classes, you either must fully qualify them as `System.Text.RegularExpressions.Regex`; or, in VB.NET, you need an `Imports` statement at the beginning of your code, as in

```
Imports System.Text.RegularExpressions.
```

Then you can test a string to determine whether it matches the regular expression using the `Match` class. Here's an example:

```
reg = New Regex(".*\.cs")
VBMacroUtilities.Print(reg.Match("hello").Success)
VBMacroUtilities.Print(reg.Match("MyFile.cs").Success)
```

The first line results in the string `False` being written to the output window, since the first string, "hello", does not match the pattern. The second string, "MyFile.cs", however, does match the pattern, and so the second line writes the string `True` to the output window.

Next is a macro that uses the previous regular expression pattern to search for all items in the Solution Explorer that end in ".cs"—that is, the macro finds all C# source files.

```
' Need
' Imports System.Text.RegularExpressions
Private reg As Regex
```

```

Sub RegTraverse(ByVal items As UIHierarchyItems)
    Dim subitem As UIHierarchyItem
    Dim m As Match
    For Each subitem In items
        m = reg.Match(subitem.Name)
        If m.Success Then
            VBMacroUtilities.Print(subitem.Name)
        End If
        If Not subitem.UIHierarchyItems Is Nothing Then
            RegTraverse(subitem.UIHierarchyItems)
        End If
    Next
End Sub

Sub DemoRegExItems()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim win As Window = DTE.Windows.Item( _
        Constants.vsWindowKindSolutionExplorer)
    Dim uih As UIHierarchy = win.Object
    reg = New Regex(".*\.cs")
    RegTraverse(uih.UIHierarchyItems)
End Sub

```

You can see how I perform the match in this code: I call the `Match` function of the `Regex` object; this function returns an object of class `Match`. This `Match` object has a member called `Success`, which is either `True` or `False`, corresponding to whether the match succeeded. If the match worked, then I print out the name of the item. Otherwise, I just move on.

Selecting a Hierarchy Item

The `UIHierarchy` object has a property called `SelectedItems`, which is an array of, as you can imagine, `UIHierarchyItem` objects that are currently selected. Now, remember, you have a `UIHierarchy` object for each window that contains a hierarchy, so if you have both the Solution Explorer and the Server Explorer windows open, each window can have a set of selected items. When you switch to one window, the other window's selected items change color to indicate they're still selected but not active. Therefore, if you are using the `UIHierarchy` object for two different windows, you might well find that both objects have a list of items in the `SelectedItems` array.



Each `UIHierarchyItem` object has a `Selected` property, meaning you have two ways of finding which items are selected: If you are traversing the hierarchy, you can simply look at an item's `Selected` property; or, if you are not traversing the hierarchy, you can obtain the list of selected items using the root `UIHierarchy` object's `SelectedItems` property.

Here's a simple macro that lists all the items selected in both the Solution Explorer and the Server Explorer windows:

```

Sub ListSelectedItems()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim win As Window = DTE.Windows.Item( _
        Constants.vsWindowKindSolutionExplorer)
    Dim uih As UIHierarchy = win.Object
    Dim uihitem As UIHierarchyItem
    VBMacroUtilities.Print("Solution Explorer")
    For Each uihitem In uih.SelectedItems
        VBMacroUtilities.Print("  " & uihitem.Name)
    Next
    win = DTE.Windows.Item( _
        Constants.vsWindowKindServerExplorer)
    uih = win.Object
    VBMacroUtilities.Print("Server Explorer")
    For Each uihitem In uih.SelectedItems
        VBMacroUtilities.Print("  " & uihitem.Name)
    Next
End Sub

```

And here's a sample output listing from this macro:

```

Solution Explorer
  AssemblyInfo.cs
  Form1.cs
Server Explorer
  book_id
  bookauth

```

Note: For the Server Explorer window, I had a books database open, and I had clicked on two fields in the list, resulting in the `book_id` and `bookauth` items in this output.



Just as you can traverse a window's hierarchy even if the window is closed, you can also obtain a list of selected items. For example, if you select two items in the Solution Explorer window and then close the window, you would still see the selected items listed in the Solution Explorer's `UIHierarchy.SelectedItems` property. IDE users might not understand this point and might get confused if you run an add-in or other program based on a selected item when the window containing the selected item is closed. I recommend, therefore, that you first check the `Window` object's `Visible` property, in addition to checking for selected items in the hierarchy.

When you are traversing a hierarchy, you can cause the IDE to behave as if the user double-clicked an item in the hierarchy. For example, if you have the `UIHierarchyItem` for a folder in the Solution Explorer, you can call the item's `Select` method. If the folder is currently expanded, then the folder will collapse. But if the folder is collapsed, then it will expand.



When you use the `UIHierarchyItem.Select` method, the focus does not move to the window containing the `UIHierarchyItem`. Instead, the focus will either remain on whichever window previously had the focus or, in the case of double-clicking a macro's name in the Macro Explorer, the focus will return to whichever window previously had the focus. If you want to set the focus to the window containing the item, call the `Window` object's `Activate` method.

Collapsing Nodes

Using a combination of the `Document` object, the `Project` object, and the various hierarchy objects, you can write a macro that will collapse any project nodes in the Solution Explorer that have no open documents. For example, if you have a solution with 10 projects, with only one source file open, but the tree in the Solution Explorer has several projects expanded (making for a somewhat messy view), then the following macro will collapse all the nodes in the tree except for the project containing the single file that is open.

```
Sub CollapseUnused()
    ' Gather all projects
    Dim openprojects As New Collection()
    Dim myproject As Project
    For Each myproject In DTE.Solution.Projects
        openprojects.Add(myproject, myproject.FullName)
    Next

    ' Remove projects from list that
    ' have an open document
    Dim doc As Document
    Dim proj As Project
    For Each doc In DTE.Documents
        Try
            proj = doc.ProjectItem.ContainingProject
            openprojects.Remove(proj.FullName)
        Catch
        End Try
    Next

    ' Close the projects in the list
    Dim win As Window = DTE.Windows.Item( _
        Constants.vsWindowKindSolutionExplorer)
    Dim uih As UIHierarchy = win.Object
    Dim item As UIHierarchyItem
    Dim projitem As ProjectItem
    Try
        For Each item In uih.UIHierarchyItems.Item(1).UIHierarchyItems
            Dim name As String = TypeName(item.Object)
            If name = "Project" Then
```

```

        proj = item.Object
    Try
        If Not openprojects.Item(proj.FullName) _
            Is Nothing Then
            item.UIHierarchyItems.Expanded = False
        End If
    Catch
    End Try
End If
Next
Catch
End Try

End Sub

```

This macro is rather straightforward: It first gathers up all the projects into a single `Collection` object, which is a general-purpose object for storing collections of items. Next, it iterates through the `DTE.Documents` list, obtaining the `ProjectItem` object for each `Document` object, and then the associated `Project` object. The code then removes the `Project` object from the collection. After the loop is finished, the `Collection` object will contain a list of all the projects in the solution that have no open documents.

In the third part of the macro, the code iterates through the list of projects. One interesting caveat to this code, however: When I extracted the `UIHierarchy` object for the Solution Explorer window, I noticed that the object contained only one entry, which corresponded to the top node in the tree, which is the solution itself. To get the projects, I therefore had to iterate through the `UIHierarchyItems` collection of that single entry, not the `UIHierarchyItems` collection of the `UIHierarchy` object. And just to make sure everything went smoothly, I put a try block around the code.

For each iteration, then, I grab the `UIHierarchyItem` object's `Object` property, and I check the type name. It should be "Project", but I do a comparison, just in case. Then I try to find the project in the `Collection` object. To find an item in a `Collection`, you call the `Item` method, passing the key, in this case the project's full name. Although I'm checking the results of `Item` against the `Nothing` value, in actuality the `Item` method will throw an exception if the project is not present. However, I don't need the results of the `Item` call; I simply need to know if the project is present. Therefore, I wrap the if statement in a `Try/Catch` block.

Moving Forward

In the next chapter I expand on the information introduced in this chapter and show you how you can modify source code, taking into consideration the actual source code elements. It will be as if your macro or add-in is aware of not only the text in the document but the way the text represents code in a particular language as well. At the end of Chapter 11, I also return to add-ins and demonstrate how you can use the information in both these chapters to build an add-in that can easily modify your code.

The CodeModel and Build Objects

In the previous chapter, “Programming the Document and User Interface Objects,” I described how you could modify the code inside the code editor by working directly with the text. However, in the examples in Chapter 10, there was really no way to determine what the different lines of code meant from a syntax point of view: Were you looking at a class? Or perhaps a function? A variable declaration?

Using some of the good old techniques I learned back in school on how to parse a file, I could show you how to figure out what the different code elements are; but, fortunately, I don’t have to: the .NET framework already has classes that can do that for me. These classes have names such as `CodeModel` and `CodeElement`, and these different code-smart classes are the topic of this chapter.



Since the three main languages available for programming in .NET are C++.NET, VB.NET, and C#, the various code model classes I discuss in this chapter support only these three languages. Further, bear in mind that, like other language-specific objects in .NET, the framework has two sets of classes: one for C++.NET and one for both VB.NET and C#. Finally, be aware that you cannot use the code model classes to modify VB.NET source code; such source code is considered read-only by the code model classes. You can, however, modify C++.NET and C# source code.

Manipulating Code with the CodeModel

Two classes exist in the .NET framework that are the primary entry points into the code model. These are:

CodeModel. This object is accessible through the `Project` object's `CodeModel` property, and therefore is associated with an entire project.

FileCodeModel. This object is accessible through the `ProjectItem` object's `FileCodeModel` property, and therefore is associated with a single file within a project.



Although the CodeModel class has members such as AddClass and AddFunction, at the time of this writing, the current version of Visual Studio. Net (the first version) does not yet support these functions for C#; however, it does for C++.

The `CodeModel` and `FileCodeModel` classes each contain a `CodeElements` property. This is a collection of code element objects that describe the code within a given file. The entire code system for a file is described hierarchically, which makes sense: In the outermost layer you will have a set of elements, such as a namespace or a class; then inside the namespaces and classes you will have members, such as variables and functions; and inside the functions you will have more variables. Each of these items—the namespaces, the classes, the variables, the functions—is a code element.

The code model system uses a separate class for each element, each of which is derived from a base class called `CodeElement`. Therefore, you can either list each element as a `CodeElement` object or as its own class. The classes have names such as `CodeClass`, `CodeNamespace`, and `CodeFunction`.



Technically speaking, the individual code element classes are not derived from CodeElement; rather, they provide the same interface as CodeElement. The CodeElement class is actually a separate COM class from the individual code element classes. However, the .NET framework lets you treat them polymorphically.

Here are some of the more useful `CodeElement` classes:

CodeClass Object. Specifies a class.

CodeEnum Object. Specifies an enumeration.

CodeFunction Object. Specifies a function.

CodeNamespace Object. Specifies a namespace.

CodeProperty Object. Specifies a property.

CodeVariable Object. Specifies a variable.



If you want to see an entire list of class names, you can find one by opening the online help and going to the CodeElement Object entry. From there, click the Object Properties, Methods, and Events link at the bottom of the entry. Scroll down and click the Kind property. The Kind property's entry has a list of class names at the bottom of the page.

To obtain a CodeModel object, start with a Project object and access its CodeModel object. Remember, a CodeModel object applies to the entire project. To obtain a FileCodeModel object, start with a ProjectItem and access its FileCodeModel property.

The CodeModel and FileCodeModel objects both have a CodeElements property. Similarly, each CodeElement object also has a CodeElements property. Since the code elements are stored in a hierarchical manner, a recursive algorithm works best to traverse through the elements. Here's a macro that climbs through all the items in the current document. First is the recursive subroutine, followed by the macro, called TestCodeModel1.

```
Sub DumpElements(ByVal indent As Integer, ByVal ces As CodeElements)
    Dim child As Object
    For Each child In ces
        Try
            VBMacroUtilities.Print("".PadLeft(indent * 4) & _
                child.Name.PadRight(45 - indent * 4) & _
                " " & TypeName(child))
            If Not child.Members Is Nothing Then
                DumpElements(indent + 1, child.Members)
            End If
        Catch
        End Try
    Next
End Sub

Sub TestCodeModel1()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim fc As FileCodeModel =
    DTE.ActiveDocument.ProjectItem.FileCodeModel
    DumpElements(0, fc.CodeElements)
End Sub
```

As usual, this macro makes use of the VBMacroUtilities assembly covered in Chapter 3, "Introducing the Visual Studio Macro IDE." The macro obtains the current document, and from that the associated project item, and finally the FileCodeModel object. The FileCodeModel object contains a CodeElements collection, which itself contains a list of CodeElement objects. It is this CodeElements collection that I pass to the recursive subroutine called DumpElements.

The recursive DumpElements subroutine takes two parameters, an indentation level, which allows me to print out the hierarchy with a respective indentations, and a CodeElements collection. (I use the PadLeft function to indent the lines that I print out.)

For each `CodeElement` object in the `CodeElements` list, `DumpElements` prints out the name of the elements and the type name. The reason I print out the type name is so you can see that even though each item in the collection is a `CodeElement` item, each has its own class name, such as `CodeClass` or `CodeFunction` or `CodeNamespace`. (They all start with the word “Code.”)

This macro uses a little trick: Not all the `CodeElement` classes have a member called `Members`. Thus, the line that calls `DumpElements` recursively, passing `child.Members`, will not always be valid. So I put a `Try/Catch` block around the loop. If the `Members` member is not present, the empty catch block will step in and the loop will continue along as if nothing had happened. (Some people have a problem with this kind of code—that is, using an exception handler as a general-purpose mechanism rather than specifically for error conditions—but in my opinion, this makes for simpler code.)

To try out this macro, open up a source code file. It can be a VB.NET file, a C# file, or a C++ file. If you use a C++ file, you will see that each type name will be preceded by “VC,” as in `VCCodeClass` or `VCCodeFunction` or `VCCodeNamespace`. The reason for the name difference is that the .NET framework maintains two different processors, one for the VB.NET and C# languages and one for the C++.NET language. However, for each class in one set, there’s a corresponding class in the other set, and the corresponding classes work similarly.

If you read Chapter 10, you are familiar with the `EditPoint` object. Each `CodeElement` class includes a `StartPoint` and an `EndPoint` object, which specify where the element begins and ends, respectively, in the code document. This way, given a `CodeElement` object, you can position the insertion point at either the beginning or ending of the code element’s position in the code; or you can highlight the entire code element.

The following macro is an enhancement to the `TestCodeModel` macro shown earlier in this section; it prints the elements and shows the starting and ending line numbers of each element as well:

```
Sub DumpElements2(ByVal indent As Integer, ByVal ces As CodeElements)
    Dim child As Object
    For Each child In ces
        Try
            Dim sp As TextPoint
            Dim ep As TextPoint
            sp = child.StartPoint
            ep = child.EndPoint
            VBMacroUtilities.Print("."PadLeft(indent * 4) & _
                child.Name.PadRight(45 - indent * 4) & _
                " " & TypeName(child) & " " & _
                sp.Line & "-" & ep.Line)
            If Not child.Members Is Nothing Then
                DumpElements2(indent + 1, child.Members)
            End If
        Catch
        End Try
    Next
End Sub
```

```

Sub TestCodeModel2()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim fc As FileCodeModel =
DTE.ActiveDocument.ProjectItem.FileCodeModel
    DumpElements2(0, fc.CodeElements)
End Sub

```

A FileCodeModel Add-in

The best way to explore how you can use the various code model classes to access and even modify the code is by seeing a real-live add-in that demonstrates some of the functionality. Therefore, in this subsection, I guide you through putting together an add-in that creates a tool window that contains a listbox. The listbox displays all the classes in a document. When you activate a different document, the listbox updates with the different document's classes. The tool window also has two buttons, one that refreshes the listbox (which you use the first time the add-in's tool window opens) and one that lets you create a new class.

But before we begin, I want to point out that the process of creating a class proved somewhat interesting when I put this together because of the following issues:

- VB.NET is read-only. Therefore, the add-in cannot modify VB.NET code; it can only show information about the code.
- C# worked as expected.
- C++.NET uses its own classes that do not behave as you might expect; therefore the code for modifying C++.NET code differs from that for modifying the C# code.

Now let's get started. Begin by creating a new add-in in VB.NET. (I used VB.NET for this one because I was able to first prototype some of the functionality as a macro, which requires VB.NET. Then when I had the parts ready to go, by making my add-in also in VB.NET, I was able to simply paste in the code from the macros.) When you create the add-in, check the box to create a toolbar menu.

When Visual Studio .NET creates the project for you, add references to Microsoft.VisualStudio.VCCodeModel.dll and the VSUserControlHostLib library.



The VSUserControlHostLib is a library that you can download from Microsoft's Web site. I provided instructions for obtaining and building the library in Chapter 7, "Creating Add-ins for the IDE," in the section "Using the Form Designer with a Tool Window."

Here is the first portion of the Connect.vb file. You can see that my class is derived from both `IDTExtensibility2` and `IDTCommandTarget`, meaning that not only is this an add-in, but it supports commands as well. You can also see I added code to the `OnDisconnection` handler that hides the tool window. Further, I removed the `applicationInstance` variable and replaced it with a global variable that I called

DTE (the DTE variable is defined later in the ClassCommands.vb module). That way, since my code started out in the macro editor, I can easily paste in the code directly, without having to replace each instance of DTE with applicationInstance.

```
' Connect.vb
Imports Microsoft.Office.Core
Imports Extensibility
Imports System.Runtime.InteropServices
Imports EnvDTE

<GuidAttribute("0F58522C-A2A6-42FE-9355-55761D369C4E"), _
ProgIdAttribute("ClassManager.Connect")> _
Public Class Connect

    Implements Extensibility.IDTExtensibility2
    Implements IDTCommandTarget

    Dim addInInstance As EnvDTE.AddIn

    Private doc As VSUserControlHostLib.IVSUserControlHostCtl = Nothing
    Private toolwin As Window = Nothing

    Public Sub OnBeginShutdown(ByRef custom As System.Array) _
        Implements Extensibility.IDTExtensibility2.OnBeginShutdown
    End Sub

    Public Sub OnAddInsUpdate(ByRef custom As System.Array) _
        Implements Extensibility.IDTExtensibility2.OnAddInsUpdate
    End Sub

    Public Sub OnStartupComplete(ByRef custom As System.Array) _
        Implements Extensibility.IDTExtensibility2.OnStartupComplete
    End Sub

    Public Sub OnDisconnection(ByVal RemoveMode As Extensibility. _
        ext_DisconnectMode, ByRef custom As System.Array) _
        Implements Extensibility.IDTExtensibility2.OnDisconnection
        toolwin.Visible = False
    End Sub
```

Next is the OnConnection handler. Very little of the original default code is left here. I first store the DTE variable; then I create the tool window, as I described in Chapter 7. Next I call SetupEvents, which is a global function in the ClassCommands.vb module. (I'll explain how it works when I discuss that module later in this section.) After setting up the events, I set up the commands and menus as I did in many of the add-ins earlier in this book.

```
Public Sub OnConnection(ByVal application As Object, _
    ByVal connectMode As Extensibility.ext_ConnectMode, _
```

```

ByVal addInInst As Object, ByVal custom As System.Array) _
Implements Extensibility.IDTExtensibility2.OnConnection

    DTE = CType(application, EnvDTE.DTE) ' This is in Commands.vb
    addInInstance = CType(addInInst, EnvDTE.AddIn)
    Dim tempdoc As Object

    Dim newguid As String = "{1021AE39-169C-46d2-BA00-BE69EE944EC3}"
    toolwin = DTE.Windows.CreateToolWindow( _
        addInInstance, _
        "VSUserControlHost.VSUserControlHostCtl", _
        "Form Host", newguid, tempdoc)
    toolwin.Visible = True
    doc = CType(tempdoc, VSUserControlHostLib.IVSUserControlHostCtl)
    Dim asm As System.Reflection.Assembly
    asm = System.Reflection.Assembly.GetExecutingAssembly()
    doc.HostUserControl(asm.Location, _
        "ClassManager.ClassManagerForm")

    SetupEvents()

    Try
        Dim commands As Commands = DTE.Commands
        Dim command1 As Command = commands.AddNamedCommand( _
            addInInstance, _
            "Show", "Class Manager", "Shows the Tool Window", True,
            59, Nothing, _
            vsCommandStatus.vsCommandStatusSupported + _
            vsCommandStatus.vsCommandStatusEnabled)
        Dim viewMenu As CommandBarPopup = _
            DTE.CommandBars("MenuBar"). _
            Controls("&View")
        Dim viewMenuBar As CommandBar = viewMenu.CommandBar
        Dim othersMenu As CommandBarPopup = _
            viewMenu.Controls("Oth&er Windows")
        Dim othersBar As CommandBar = othersMenu.CommandBar
        command1.AddControl(othersBar, 1)

    Catch
    End Try
End Sub

```

The `Exec` and `QueryStatus` functions follow. The only command that this add-in supports is the `Show` command, which works in conjunction with the menu item for displaying the tool window. This is the same as described in Chapter 7 as well.

```

Public Sub Exec(ByVal cmdName As String, ByVal executeOption As _
vsCommandExecOption, ByVal varIn As Object, _

```

```

ByRef varOut As Object, ByRef handled As Boolean) _
Implements IDTCommandTarget.Exec
    handled = False
    If (executeOption = _
        vsCommandExecOption.vsCommandExecOptionDoDefault) Then
        If cmdName = "ClassManager.Connect.Show" Then
            toolwin.Visible = True
            handled = True
            Exit Sub
        End If
    End If
End Sub

Public Sub QueryStatus(ByVal cmdName As String, _
    ByVal neededText As vsCommandStatusTextWanted, _
    ByRef statusOption As vsCommandStatus, _
    ByRef commandText As Object) _
Implements IDTCommandTarget.QueryStatus
    If neededText = EnvDTE.vsCommandStatusTextWanted. _
        vsCommandStatusTextWantedNone Then
        If cmdName = "ClassManager.Connect.Show" Then
            statusOption = CType(vsCommandStatus. _
                vsCommandStatusEnabled + vsCommandStatus. _
                vsCommandStatusSupported, vsCommandStatus)
        Else
            statusOption = vsCommandStatus. _
                vsCommandStatusUnsupported
        End If
    End If
End Sub

End Class

```

Next is the `ClassManagerForm.vb`. Creating this form involves the same process as described in Chapter 7: right-click on the project name, and in the popup menu choose Add➔Add User Control. Call the form `ClassManagerForm` (even though it's technically not a form). When the user control opens, you will build it as shown in Figure 11.1; place a large listbox on the left side and two buttons on the right. Then set the following: the listbox's `Dock` property to `Left`; the `Text` property of the top button to `Add Class`, and its `Name` property to `AddClassButton`; the `Text` property of the bottom button to `Refresh`, and its `Name` property to `RefreshButton`. Then right-click on the form and choose `View Code`. The code for the form will open; expand the region called `Windows Form Designer-generated code` by clicking on the small plus sign. Then add the line shown inside the constructor (the `New` subroutine)—the line `ManagerForm = Me`. This line is needed because the code in the third module, `ClassCommands.vb`, will be accessing the `ClassManagerForm` object; the `ManagerForm` variable is in the `ClassCommands.vb` module.

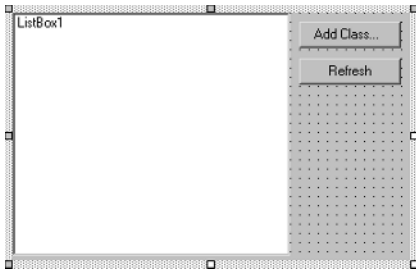


Figure 11.1 The ClassManagerForm.

```
' Remember to add a reference to
' Microsoft.VisualStudio.VCCodeModel!
' Also, added these two imports lines
Imports EnvDTE
Imports Microsoft.VisualStudio.VCCodeModel

Public Class ClassManagerForm
    Inherits System.Windows.Forms.UserControl

    #Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

        ' Added by Jeff
        ManagerForm = Me

    End Sub

    'UserControl overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As
Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.IContainer
```

```
'NOTE: The following procedure is required by the
'Windows Form Designer
'It can be modified using the Windows Form Designer.
'Do not modify it using the code editor.
Friend WithEvents ListBox1 As System.Windows.Forms.ListBox
Friend WithEvents AddClassButton As System.Windows.Forms.Button
Friend WithEvents RefreshButton As System.Windows.Forms.Button
<System.Diagnostics.DebuggerStepThrough()> Private Sub _
InitializeComponent()
    Me.ListBox1 = New System.Windows.Forms.ListBox()
    Me.AddClassButton = New System.Windows.Forms.Button()
    Me.RefreshButton = New System.Windows.Forms.Button()
    Me.SuspendLayout()
    '
    'ListBox1
    '
    Me.ListBox1.Dock = System.Windows.Forms.DockStyle.Left
    Me.ListBox1.Name = "ListBox1"
    Me.ListBox1.Size = New System.Drawing.Size(240, 212)
    Me.ListBox1.TabIndex = 0
    '
    'AddClassButton
    '
    Me.AddClassButton.Location = New System.Drawing.Point(248, 8)
    Me.AddClassButton.Name = "AddClassButton"
    Me.AddClassButton.Size = New System.Drawing.Size(88, 23)
    Me.AddClassButton.TabIndex = 1
    Me.AddClassButton.Text = "Add Class..."
    '
    'RefreshButton
    '
    Me.RefreshButton.Location = New System.Drawing.Point(248, 40)
    Me.RefreshButton.Name = "RefreshButton"
    Me.RefreshButton.Size = New System.Drawing.Size(88, 23)
    Me.RefreshButton.TabIndex = 2
    Me.RefreshButton.Text = "Refresh"
    '
    'ClassManagerForm
    '
    Me.Controls.AddRange(New System.Windows.Forms.Control() _
        {Me.RefreshButton, Me.AddClassButton, Me.ListBox1})
    Me.Name = "ClassManagerForm"
    Me.Size = New System.Drawing.Size(344, 216)
    Me.ResumeLayout(False)

End Sub

#End Region
```

Next come the form's member functions. The Run function is the entry function that calls a recursive function that climbs through the code elements, looking strictly for the class elements, adding each one to the listbox. The handler for the Refresh button follows, which calls a RefreshVars function (which gets the latest Document object) and then calls the Run subroutine. The handler for the Add Class button follows. This handle simply calls the AddClass subroutine, which is part of the next module, ClassCommands.vb. The subroutine adds a class to the code, thereby actually modifying the code. Finally, the handler for the listbox control's double-click event does some sanity checks to make sure all is fine before calling HighlightClass, which is in the ClassCommands.vb module, which I describe shortly.

```

Sub Run()
    If DTE.ActiveDocument Is Nothing Then
        Exit Sub
    End If
    ListBox1.Items.Clear()
    RecurseExtract(0, RootElement.CodeElements)
End Sub

Sub RecurseExtract(ByVal indent As Integer, ByVal ces As _
EnvDTE.CodeElements)
    Dim child As Object
    For Each child In ces
        Try
            If TypeOf (child) Is EnvDTE.CodeClass Then
                ListBox1.Items.Add(New AddinCodeClass(child, 0))
            End If
            If Not child.Members Is Nothing Then
                RecurseExtract(indent + 1, child.Members)
            End If
        Catch
        End Try
    Next
End Sub

Private Sub AddClassButton_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles AddClassButton.Click
    AddClass()
End Sub

Private Sub RefreshButton_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles RefreshButton.Click
    UpdateVars()
    Run()
End Sub

Private Sub ListBox1_DoubleClick(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles ListBox1.DoubleClick
    If CurrentDocument Is Nothing Then

```

```

        Exit Sub
    End If
    Dim index As Integer = ListBox1.SelectedIndex
    If index = -1 Then
        Exit Sub
    End If
    Try
        CurrentDocument.Activate()
        Dim obj As Object = ListBox1.Items(index).codeinst
        HighlightClass(obj)
    Catch
    End Try
End Sub

End Class

```

Now comes the `ClassCommands.vb` module. This module contains the work portions of the add-in. Most people agree good programming practice includes separating the user interface from the processing. However, in reality that's easier said than done. I did break away the processing from the interface in this program, with one main exception: I stored the list of class objects in the listbox itself. I suppose if I were more adventurous, I could subclass the `Listbox` class and override its protected `CreateItemCollection` method, providing access to my own collection that I would store in the processing module called `ClassCommands`. But why add that kind of complexity? If this were a large, production-scale project, perhaps that would be a good idea, but for a small project such as this, I say leave well enough alone.

You can see that this module includes a class called `AddinCodeClass`. This class is simply a wrapper around the `CodeClass` class. The primary purpose for this class is to provide a `Tostring` function that writes out the full name of the code element, with an indentation level. Earlier, in the `RecurseExtract` function of the `ClassManagerForm.vb` module, I populate the `Listbox` control with instances of `AddinCodeClass`. The `Listbox` class includes a feature whereby you can fill the control with instances of an object; then, for each line in the list, the `Listbox` class calls the object's `Tostring` function. Thus, when I fill the `Listbox` control with instances of `AddinCodeClass`, the `FullName` of the element will display in the listbox. Further, when the user double-clicks a line in the `Listbox` control, I'm able to immediately extract the object associated with the line the user clicked.

This module also contains the `AddClass` subroutine. This subroutine first makes sure the various variables have legitimate data in them, then it checks which programming language the current document is in. If the language is C++ (that is, the `FileCodeModel.Language` property is `vsCMLanguageVC`), then the code adds a class using the `FileCodeModel` object's `AddClass` function.

If the language is C#, then the code tries to add the class at the innermost position possible, based on where the current insertion point is. (This functionality does not work for C++, only C#, thus the difference in code based on the language.) If the innermost element does not have an `AddClass` function (in which case an exception will be thrown), then the code will look to the parent element, trying again.



You might be wondering why I chose to use an exception handler to determine if the element has an `AddClass` function, rather than simply using the runtime type information, which is available through the .NET framework's `Type` class. The reason is that although the `TypeName` function returns the correct class name for COM objects stored in an `Object` variable, the `GetType` function simply returns a COM object, with no knowledge of the underlying class. Therefore, I could not make use of the `Type` class; it always responded that there was no `AddClass` function even when there was.

```
'ClassCommands.vb
Imports EnvDTE
Imports Microsoft.VisualStudio.VCCodeModel

Module ClassCommands

    Public DTE As EnvDTE.DTE

    Public Class AddinCodeClass
        Public codeinst As Object
        Public indent As Integer
        Public Overrides Function ToString() As String
            Return "".PadLeft(indent * 4) & codeinst.FullName
        End Function
        Public Sub New(ByRef acodeinst As Object, _
            ByVal aindent As Integer)
            codeinst = acodeinst
            indent = aindent
        End Sub
    End Class

    Public CurrentDocument As Document = Nothing
    Public RootElement As EnvDTE.FileCodeModel = Nothing
    Public ManagerForm As ClassManagerForm = _
        Nothing ' Set in ClassManagerForm constructor

    Sub UpdateVars()
        CurrentDocument = DTE.ActiveDocument
        RootElement = DTE.ActiveDocument.ProjectItem.FileCodeModel
    End Sub

    Sub HighlightClass(ByVal CodeInst As Object)
        Dim sel As TextSelection
        sel = CurrentDocument.Selection
        sel.MoveToPoint(CodeInst.StartPoint, False)
        sel.StartOfLine(vsStartOfLineOptions. _
            vsStartOfLineOptionsFirstColumn)
        sel.MoveToPoint(CodeInst.EndPoint, True)
    End Sub
```



```
Sub AddClass()  
    ' Obtain the current position in the  
    ' code and insert the class there  
    If CurrentDocument Is Nothing Then  
        Exit Sub  
    End If  
    If Not TypeOf (CurrentDocument.Object) Is TextDocument Then  
        Exit Sub  
    End If  
    Dim tdoc As TextDocument = CurrentDocument.Object  
  
    ' Find out the language  
    Dim lang As String = CurrentDocument.ProjectItem. _  
        FileCodeModel.Language  
  
    Dim newclass As String  
    ' The online help fails to mention CodeModelLanguageConstants...  
    If lang = EnvDTE.CodeModelLanguageConstants.vsCMLanguageVC Then  
        newclass = InputBox("Enter the class name")  
        Try  
            Dim vcm As VCFileCodeModel = DTE.ActiveDocument. _  
                ProjectItem.FileCodeModel  
            vcm.AddClass(newclass)  
        Catch ex As Exception  
            MsgBox(ex.Message)  
        End Try  
    ElseIf lang = EnvDTE.CodeModelLanguageConstants. _  
        vsCMLanguageCSharp Then  
        newclass = InputBox("Enter the class name")  
        Try  
            Dim element As Object  
            Dim done As Boolean = False  
            Dim tname As String  
            element = tdoc.Selection.ActivePoint.CodeElement( _  
                vsCMElement.vscMEElementModule)  
            While Not done  
                Try  
                    'tname = TypeName(element)  
                    ' Can't use typeinfo(element) and then  
                    ' GetMethod -- see text  
                    'MsgBox("Trying class " & tname)  
                    element.AddClass(newclass)  
                    done = True  
                Catch  
                    element = element.Collection.Parent  
                End Try  
            End While  
        Catch ex As Exception  
            MsgBox(ex.Message)
```

```

        End Try

    End If

End Sub

' Event Handlers
Dim WithEvents winevents As WindowEvents

Sub SetupEvents()
    winevents = CType(DTE.Events.WindowEvents, EnvDTE.WindowEvents)
End Sub

Private Sub WindowEvent(ByVal GotFocus As Window, _
    ByVal LostFocus As Window) Handles winevents.WindowActivated
    If GotFocus.Kind = "Document" Then
        ' Can't call UpdateVars at this point, since
        ' DTE.ActiveDocument isn't set yet
        CurrentDocument = GotFocus.Document
        RootElement = GotFocus.Document.ProjectItem.FileCodeModel
        ManagerForm.Run()
    End If
End Sub

End Module

```

After you compile this add-in, the first time you run it, it will install a menu item, **View→Other Windows→Class Manager**. In future runs of the add-in, you can use this menu item to launch the add-in. You can also use this menu item to display the tool window again after you close it. The first time you run the add-in, the tool window will open automatically, as shown in Figure 11.2. The tool window contains the controls you drew in the user control form. On the left is a listbox that will contain the names of the classes in a code document. To see the classes, make sure you have a code file open. If you already do, and you just opened the add-in tool window, click the tool window's Refresh button; or simply click on the document. The listbox will fill with the names of the classes in the current document.

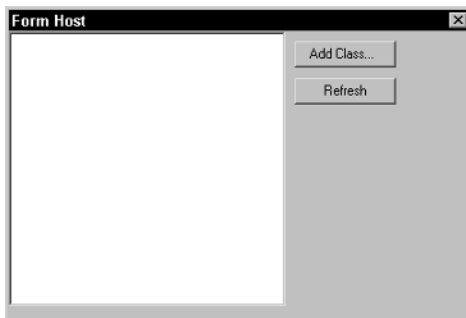


Figure 11.2 The add-in opens a tool window.

You can then click the Add Class button to add a class to the document. The code will insert the class in a logical place inside the code, formatted for either C# or C++.



Remember, you cannot use an add-in to modify VB.NET code using the code model; to modify VB.NET code, you need to manually write your VB.NET code to the document using the methods I described in Chapter 10.

If you're interested in exploring the code model further, you might start by enhancing this Class Manager add-in. Here are some things to consider doing:

- Modify the code so that when it loads it automatically fills the listbox with the classes for the active document, without you having to first click the Refresh button.
- Add buttons for Add Function and Add Variable.
- Request the name, type, and parameters for your Add Function button. Take a look at the online help for the `AddFunction` method to see how to specify these. (Note: The type can be a string or an instance of the `Type` class.)
- Allow the user to specify additional information such as base classes for the Add Class button. Take a look at the online help for the `AddClass` method to see how.

Working with Build Objects

The automation model contains classes that let you interact with the IDE's build process. The main object you use is the `SolutionBuild` object, which includes functions that let you: start a build of the entire solution or just a single project, start up a debug session, or run a program. You can obtain the `SolutionBuild` object from the `Solution` object's `SolutionBuild` property. Thus, you use `DTE.Solution.SolutionBuild`.

To build an entire solution, use the `SolutionBuild.Build` method, which is equivalent to choosing `Build⇨Build Solution`. To build a particular project, use `SolutionBuild.BuildProject`. (`SolutionBuild.BuildProject` doesn't really have an equivalent menu item, since the function can build any project in the solution, whereas the `Build Project` section of the `Build` menu builds only the active project.)



When you build a solution or project from a macro, you pass either `True` or `False` as a parameter, `True` meaning the call will pause and not return until the build is complete. Be careful when you pass `True`, however, because you really cannot interrupt the macro once the build has started, nor can you interrupt the build. The IDE freezes as it normally does when you run a macro, meaning you can't halt the build. And if you abort the macro by double-clicking the macro icon in the system tray, the macro won't stop until the build finishes. So use care when passing `True` to the build functions.

Here's an example. Suppose you have two related solutions, and while working in one, you occasionally have to build the other, but you don't want to interrupt your current work by closing the present solution, loading the other, building it, closing it, and returning to the first. I'll call the present solution Solution One, and the solution that is not loaded Solution Two. Now suppose that Solution Two contains projects whose files live inside source code control and that other people can change the files. That means that even though you may not have made any changes to Solution Two, your build of it could be outdated. So before you complete your build of Solution One, you want to go ahead and build Solution Two.

To build Solution Two, you could start up a second instance of Visual Studio .NET; the IDE allows you to do so. But if you're not going to be modifying code and working on the projects in Solution Two, why mess around with two instances of the IDE? Visual Studio .NET provides you with three ways you can build Solution Two:

- Open up a second instance of the IDE, even though I just suggested otherwise. (You might be fine with having two instances running. Personally, I find it cumbersome.)
- Open a command-line prompt (i.e., DOS window) and run the command-line version of the Visual Studio .NET product.
- Invoke a macro that builds the solution for you.

The first of these, starting a second instance of the IDE, is pretty self-explanatory: You just run Visual Studio .NET again. The second is quite simple, too, provided you open a DOS window that has the paths and other environment variables set properly. (The Start menu item in Microsoft Visual Studio .NET has a submenu called Visual Studio .NET Tools that includes a Visual Studio .NET Command Prompt item, which I suggest you use for opening your DOS window. It has all the path and other environment variables set up properly for you.)

As I mentioned in Chapter 7, "Creating Add-ins for the IDE," you can invoke the Visual Studio .NET program from the command prompt. Here's the DOS command for invoking `devenv` in command-line mode to build your solution for you:

```
devenv c:\dev\SolutionTwo\SolutionTwo.sln /build DEBUG
```

Easy enough. But if you want to automate the build, you can use a macro. Your macro can take three different approaches:

- It could launch a DOS window and execute the `devenv` command.
- It could temporarily unload Solution One, load Solution Two, build it, unload it, and then reload Solution One.
- It could launch another instance of the IDE, build the solution, and then close.

Each of these is reasonable, and the one you choose will depend simply on personal taste. I kind of like the high-tech automation approach to unloading my current solution, loading and building another, and then returning to the first, although it might seem like a bit of overkill. Nevertheless, I'll show you macros for each of these three possibilities in the following two sections.

You might even be creative enough to come up with other approaches I haven't considered. Remember, however, that the IDE is only capable of having one solution open at a time. I suppose it's possible to conceive of an IDE capable of managing multiple solutions simultaneously, but for the majority of the users out there, such a feature would probably cause more complications than it would be worth. Or, perhaps if you need to build multiple solutions on a regular basis, you might write an add-in that includes a tool window that holds a list of other solutions that you can edit. You could then double-click one of the solution names, after which the add-in would spawn a build process in the same way I describe in the next section. All that said, I'm not going to present an add-in here that lists a group of solutions, as I doubt many readers would have much use for it.

Spawning a Build Process

To spawn a program from a macro, you can use Visual Basic's built-in Shell command. Like `MsgBox` and `InputQuery`, `Shell` is not a part of the .NET framework; instead, it is a keyword in the Visual Basic language.

Alternatively, you can tell Visual Studio .NET to spawn a command by issuing the `Tools.Shell` command. Remember, the IDE maintains a list of commands (of which your macros are a part, as are the commands you add to your add-ins). One such command is the `Shell` command, which is part of the `Tools` command namespace.

Later in this section I'll show you two macros that perform these two spawn processes. But first I need to mention that in order to run the `devenv` in the command-line environment, the environment must be set up properly. Earlier I mentioned starting the command prompt that's configured for .NET. The catch is getting this configuration into the command that you spawn. Here's what I recommend you do: If you are going to be launching the `devenv` process from within the IDE, start the IDE from the .NET command prompt. But instead of just opening the .NET command prompt and typing `devenv` to run the IDE (which you could do), I suggest you take a slightly different approach. Normally, when you start the .NET command prompt, the Windows shortcut spawns a command shell, executing a particular batch file called `vsvars32.bat`. By default, the file is found here:

```
C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools
```

This batch file sets up the environment for you. When the shortcut starts the command prompt, it uses the `/k` option, which instructs the command shell to run the specified program (in this case, the batch file) and then remain open. Therefore, I suggest you create *another* batch file by copying the `vsvars32.bat` to a new name, called `startenv.bat`. Then open the `startenv` in a text editor of your choice (or in the IDE's own text editor) and go to the very bottom and add the following two lines (immediately after the line `":end"`):

```
devenv
exit
```

The first line causes the batch file to launch the Visual Studio .NET IDE; but now the IDE will inherit the path information and environment. The second line closes the temporary DOS window that the batch file was running under.

Next, you can create a shortcut, on the desktop, on the start menu, or on the Quick Launch bar. The shortcut's command line (that is, the "target") should be this:

```
%comspec% /k "c:\Program Files\Microsoft Visual Studio .NET\Common7\
Tools\startenv.bat"
```

(Note that %comspec% is simply a default environment variable for the command line interpreted. On my Windows 2000 system, it points to C:\WINNT\system32\cmd.exe.) You can leave the working directory (the "Start in" box) empty.

Now when you start Visual Studio .NET, use your new shortcut rather than the default one. Then the IDE will inherit the proper environment, as will any programs that your macros spawn.

Here, then, is a small macro that spawns the devenv command in a DOS window using the built-in Visual Basic command called Shell:

```
Sub SpawnBuild()
    Shell( _
        "cmd /k devenv "C:\dev\SolutionTwo\SolutionTwo.sln" /build DEBUG")
End Sub
```

In this command I launch the cmd command, which forces devenv, when running in command-line mode, to send its output to a DOS window. (Otherwise, you won't see any output and devenv will run in the background, leaving you unable to see the results of the build.) The /k option tells the cmd program to keep the DOS window open after the following command—in this case devenv—finishes executing. That way you can look at the output and then type exit to close the DOS window. If you prefer to have the window close right away, you can replace /k with /c.



When I run the SpawnBuild macro on my computer, the DOS window does not automatically come to the front; I have to click its icon on the task bar. However, the following macro, SpawnBuild2, does bring the DOS window to the front automatically.

Now here's a macro that also spawns devenv, but instead using the Tools.Shell command.

```
Sub SpawnBuild2()
    DTE.ExecuteCommand("Tools.Shell", _
        "cmd /k devenv "C:\dev\SolutionTwo\SolutionTwo.sln" /build DEBUG")
End Sub
```

This macro works in precisely the same way as the previous SpawnBuild macro, except it uses the IDE's Tools.Shell command to spawn the devenv program.

Building with the SolutionBuild Object

In addition to spawning a build, you can use the SolutionBuild object to build an object in the IDE itself. To obtain the SolutionBuild object, use `DTE.Solution.SolutionBuild`. Then, to build the current solution, call `Build`. Here's a simple example:

```
Sub BasicBuild()
    Dim sln As Solution = DTE.Solution
    If sln.FullName = "" Then
        MsgBox("No solution currently loaded")
    Else
        sln.SolutionBuild.Build(False)
        MsgBox("Building!")
    End If
End Sub
```

In the previous section, the other macro option I mentioned was to momentarily switch to another solution. Remember, however, that's just an example of how you might automate the process of building a solution. Here, then, is an example of a macro that will switch to a different solution and then back again:

```
Sub SwitchAndBuild()
    Dim win As Window = DTE.Windows.Item( _
        EnvDTE.Constants.vsWindowKindOutput)
    win.Activate()
    Dim ow As OutputWindow = win.Object
    Dim pane As OutputWindowPane = ow.OutputWindowPanes.Item(1)
    pane.Clear()
    pane.OutputString("Opening solution...")
    Dim sln As Solution = DTE.Solution
    Dim cursoln As String = DTE.Solution.FullName
    sln.Open("C:\dev\SolutionTwo\SolutionTwo.sln")
    sln.SolutionBuild.Build(True)
    sln.Open(cursoln)
End Sub
```



When you build a solution using the `SolutionBuild.Build` function, you will automatically use whichever configuration (e.g., Debug or Release) is currently active. The `Build` function does not give you the option to choose. However, the `BuildProject` function, which I discuss in the next section, “More on the SolutionBuild Object,” does let you specify a configuration.

More on the SolutionBuild Object

The SolutionBuild object lets you build either an entire solution or an individual project. To build an individual project, you call the `BuildProject` function. The parameters

to `BuildProject` are a bit different from other functions in the automation model, in that you don't pass a `Project` object to the function. Instead, to specify the project you wish to build, you pass the *name* of the project. But which name do you use? A `Project` object has multiple names, including `FullName` and `UniqueName`. The name to use is the `UniqueName` property.

Additionally, you pass the name of the configuration you want to build (e.g., `Debug` or `Release`). Finally, you pass a `True` if you want the function to pause and wait until the build process is finished before continuing; otherwise, you can pass `False`, which is the default.

Here's an example of a call to `BuildProject`:

```
Sub BuildProjectDemo()
    Dim proj As Project
    proj = DTE.Solution.Projects.Item(1)
    MsgBox(proj.UniqueName)
    DTE.Solution.SolutionBuild.BuildProject( _
        "Debug", proj.UniqueName, False)
End Sub
```

You can see that, oddly, the first parameter is the configuration, and then you pass the project's unique name. In this macro I included a message box that shows the unique name of the project so that you can see the format. (It's usually a local directory name, followed by a backslash, followed by the project filename, as in `myproject\myproject.vbproj`.) Also remember that the configuration name is the configuration name for the solution, not for the project.

You can also clean the projects simply by calling `SolutionBuild.Clean`, passing a `True` if you want the function to wait for the clean operation to complete, or passing `False` or leaving the parameter blank if you don't want to wait. The following macro demonstrates this:

```
Sub CleanProjects()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    VBMacroUtilities.Print("The following projects will be cleaned.")
    Dim solnctx As SolutionContext
    Dim solncfg As SolutionConfiguration
    solncfg = DTE.Solution.SolutionBuild. _
        SolutionConfigurations.Item("Debug")
    For Each solnctx In solncfg.SolutionContexts()
        If solnctx.ShouldBuild = True Then
            VBMacroUtilities.Print(solnctx.ProjectName)
        End If
    Next
    DTE.Solution.SolutionBuild.Clean()
End Sub
```

In this macro, I also demonstrate how you can discover which projects are going to be cleaned; the same list of projects will be built when you build the entire solution. The `SolutionBuild` object contains a list of `SolutionConfiguration` objects, one

for each project in the solution. The `SolutionConfiguration` object includes a `ShouldBuild` property, which specifies whether the project is included in the build or clean process. This list corresponds to the items that have checkmarks by them in the Configuration Manager dialog. You can open this dialog by right-clicking the solution name in the Solution Explorer and choosing Configuration Manager in the popup menu. The dialog box contains a list of projects and a checkbox for each under the header Build. Those projects that have a check in the Build column will have a value of `True` for the `ShouldBuild` property.

The `SolutionBuild` object includes the following functions in addition to those I've already described:

Run. This function takes no parameters; it runs the startup project.

Debug. This function also takes no parameters; it starts up the debugger, running the startup project.

Deploy. This function, which takes a `true` or `false` stating whether you want the function to wait for completion, causes the projects marked for deployment to deploy, referring to the projects whose `SolutionContext` object has the `ShouldDeploy` property set to `True`. (This is the same `SolutionContext` that contains the `ShouldBuild` property.)

Moving Forward

In this chapter I introduced you to the classes you can use for analyzing and, in the case of C++ and C#, modifying the code. I demonstrated some of the code analysis and modification features using an add-in that looks for classes and lets you add a class. And though the modification features are not available for Visual Basic, as an exercise, you might think about how you can manually modify Visual Basic code by directly editing the Document object's Selection object.

In the next chapter, I move on to the topic of creating project wizards. When you create a new project in the main IDE, the first thing you do is choose a template. The templates usually consist of wizards; using the techniques I describe in the next chapter, you can create your own additional wizards.

Creating Project Wizards

When you choose File→New→Project, the New Project dialog box opens. This dialog contains the templates for the different projects available. The dialog will have different categories depending on which products you have installed, such as just VB.NET or just C#. The templates that you see in the New Project dialog are a mixture of projects and wizards. For example, take a look at the following directory (which you'll need to adjust if you installed Visual Studio .NET in a different location on your hard drive.):

```
C:\Program Files\Microsoft Visual Studio .NET\vb7\VBProjects
```

Here are two files I see in my VBProjects directory:

```
ConsoleApplication.vsz  
EmptyProject.vbproj
```

The first file represents a wizard; the second file represents a project. If you open the project file into a text editor, you will see some XML data that represents the project. You will also see that the project contains no project items or files. Indeed, the one that I list here is called EmptyProject.vbproj. When you create a project of type Empty Project, you start out with no files, just a project.

But when you create a project of the type represented by the wizard, usually a second dialog box will open following the New Project dialog box. This second dialog box will let you choose various options regarding the project; then the wizard will create a project for you with some default files. Or, if the second dialog box doesn't open, the wizard will start up and automatically create a set of files for you.

If you open the `ConsoleApplication.vsz` wizard file into a text editor, here is what you will see:

```
VSWIZARD 6.0
Wizard=VsWizard.VsWizardEngine
Param="WIZARD_NAME = ConsoleApplication"
Param="WIZARD_UI = FALSE"
Param="PROJECT_TYPE = VBPROJ"
```

The first line is simply the version of Visual Studio .NET for which this particular wizard was written. (And note that the sample shows 6.0, which is the version that preceded .NET. Although the files that this wizard inserts into your project have changed, the wizard is written to target the 6.0 engine. In general, any wizard you create will have 7.0.)

The second line is the COM component the IDE runs to launch the wizard. This particular COM component is called `VsWizard.VsWizardEngine` (which in this chapter I call the *wizard engine*). If you look inside the Registry Editor (`regedit.exe`) and expand the `HKEY_CLASSES_ROOT` key, you can scroll down and find an entry for `VsWizard.VsWizardEngine`.

The next three lines are parameters that the IDE passes to the wizard. The parameters describe the wizard: its name, whether it displays a dialog box (UI for user interface), and the type of project the wizard creates, in this case a VB.NET project.

This particular wizard, then, is really a set of parameters and, as you'll see in the next section, a set of files that communicate with the wizard engine, `VsWizard.VsWizardEngine`.



From the preceding discussion, you can see two entries to the wizard system: one is the COM component itself and the other is by passing parameters to the `VsWizard.VsWizardEngine` COM component. Therefore, you have two ways to create a wizard: either write a COM component or provide information to the default `VsWizard.VsWizardEngine` component.

In the sections that follow, I only give you information on writing a wizard using the default `VsWizard.VsWizardEngine` component. I steer clear of the COM component method simply because you have much more flexibility when you use the wizard engine. The wizard engine includes features such as *file rendering*, which enables you to generate source code that is customized to the IDE user's selections in the wizard. If you write your own COM component from scratch, you would have to write your own text-processing routines to handle such work. But why bother? The functionality is already present, so I recommend that you simply use the default `VsWizard.VsWizardEngine` component.

Dissecting the Wizard Directory Structure

Writing a wizard that lives as a script interacting with the wizard engine involves writing a JavaScript file and, optionally, various HTML files. The HTML files represent the user

interface for the script. The reason you use HTML is at the script engine displays the wizards inside an Internet Explorer control (that is, the HTML-rendering control that is the heart of Internet Explorer).



Wizards do not have to follow the model I'm describing in this section. It's just that the wizard engine, which is simply a COM component, uses this model, so if you want to write a wizard that interacts with the wizard engine (as opposed to writing your own COM component), then you must follow this model.

Here are the files you create that interact with the wizard engine:

.vsz and .vsdir files. These files describe the wizard. I'll explain where these go in a moment, when I talk about the directory structure.

HTML files. These files represent the user interface. You will have a main file called default.htm, which the script engine initially uses, and other files with any names you prefer.

Images. These are the image files that the wizard displays in the HTML pages.

Script file. Usually written in JScript (although you can use VBScript if you prefer), this file describes the actions for creating the project, copying the files into the project directory, and setting up the project options. Normally, you only have one script file, and it is called default.js.

Templates. These are the files that the script copies into the project directory. These files have two possible variations. A template file can be a text file containing symbols that the wizard will replace (such as a symbol representing the project name, which the wizard replaces with the actual project name), or a file that is not text, which the wizard simply copies into the project directory.



Although you can write your scripts in either JScript or VBScript, all of the scripts that ship with Visual Studio .NET use JScript. Therefore, for consistency, in this book I focus strictly on JScript. If you're not familiar with JScript, its syntax is very similar to that of C++ and C#.

The wizard files for a single product (that is, C++, VB.NET, and C#) live in two different directory areas. First, each product has a projects directory containing the general information files about the wizards. This directory is where you will put your .vsz and .vsdir files. However, the name of this directory differs for each product. Additionally, other project types inside Visual Studio .NET have their own project directories as well. Here are all the ones you are likely to find on your computer:

Visual Basic project wizards. c:\Program Files\Microsoft Visual Studio .NET\Vb7\VBProjects

Deployment and Setup project wizards. c:\Program Files\Microsoft Visual Studio .NET\Common7\Tools\Deployment\VsdProjects

Database project wizards. c:\Program Files\Microsoft Visual Studio .NET\Common7\Tools\Templates\Database Projects

Add-in project wizards. c:\Program Files\Microsoft Visual Studio .NET\Common7\IDE\Extensibility Projects

C++ project wizards. c:\Program Files\Microsoft Visual Studio .NET\Vc7\VCProjects

C# project wizards. c:\Program Files\Microsoft Visual Studio .NET\VC#\CSharpProjects

Additionally, the following directory holds the single solution template: c:\Program Files\Microsoft Visual Studio .NET\Common7\IDE\SolutionTemplates.

Each of these directories contains the .vsz and .vsdir files that describe the wizards available for the particular project type. Visual Studio .NET knows where these directories are located by looking in the Registry under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.0\NewProjectTemplates\TemplateDirs key. This key contains several subkeys that define the project types you see on the left side of the New Project dialog box.



Although you might consider adding more keys to the Registry, hoping to create more project types, I don't recommend doing so, primarily because you will be in for a major undertaking. Adding project types requires the Software Development Kit (SDK) that you obtain when you join the Visual Studio Integrator Program (VSIP). For each project type in the Registry there exists a package under the Packages key, and building a package requires the VSIP. Without the VSIP, you would be in for quite a job.

In addition to the .vsz and .vsdir files in the projects directory, you will find a wizards directory under most of the installations, with some exceptions. As before, the actual name varies for each installation. Here are the directory names, along with the exceptions:

Visual Basic project wizards. c:\Program Files\Microsoft Visual Studio .NET\Vb7\VBWizards.

Deployment and Setup project wizards. The deployment and setup wizards use a separate wizard from the VsWizard.VwWizardEngine wizard engine. It is called Microsoft.VSWizards.Deploy.SetupWizard, and it finds the wizard files in the same directory as the .vsz and .vsdir files.

Database project wizards. Visual Studio .NET has only one database project wizard, and it's really just an empty project not a wizard. The empty project lives in the same directory as the .vsdir file, C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools\Templates\Database Projects.

Add-in project wizards. These wizards live in subdirectories underneath the c:\Program Files\Microsoft Visual Studio .NET\Common7\IDE\Extensibility Projects directory.

C++ project wizards. c:\Program Files\Microsoft Visual Studio .NET\Vc7\
VC#Wizards

C# project wizards. c:\Program Files\Microsoft Visual Studio .NET\VC#\
VCWizards



If you go to the wizard directories (VBWizards, VCWizards, etc.), you will see several other wizards in addition to the project wizards. These are file wizards. When you add an item to a project, you can add one of many types of files. For example, in a VB.NET project, you can add a module file or a class file, among others. Each of these file types has a wizard. The .vsz and .vsdir files for these wizards are in the VBProjectItems directory, which is also off the VB.NET main installation directory. The other installed products have similar directories.

Because of the directory layout I have described, when you create a wizard, you will put its files in two locations. First, you will put the description files (.vsz and .vsdir files) in the projects directory for the particular installation (VB.NET, C#, C++). Second, you will put the scripts and additional files (the files that make up the actual wizard) in the wizards directory for the particular installation.

Wizard File Interactions and Symbols

Before I show you how to build a wizard, I want to take a moment to explain how all the files fit together and how they communicate with one another.

To begin, take a look at an existing wizard. Using either an Explorer window or a DOS prompt, head over to the projects directory for the C++ installation, which, by default, is C:\Program Files\Microsoft Visual Studio .NET\Vc7\vcprojects. Inside this directory, notice these two files:

Win32Wiz.vsz. This is the description for the Win32 project wizard.

Win32Wiz.ico. This is the icon for the Win32 project. It appears in the New Project dialog box.

In the first section of this chapter, I showed you what's inside a .vsz file. If you open this Win32Wiz.vsz file in a text editor, such as Notepad, you will see similar information: the name of the wizard COM component (`vsWizard.VsWizardEngine`) and the name of the wizard, passed in as a parameter ("`WIZARD_NAME = Win32Wiz`").

Now head up one directory then down to the VCWizards directory. Remember, the wizard name is Win32Wiz, therefore, underneath the VCWizards directory is a subdirectory called Win32Wiz. Move into this Win32Wiz directory and you'll see four subdirectories: html, images, scripts, and templates. These are the directories I described in the previous section, "Dissecting the Wizard Directory Structure." Under each of these directories, in turn, is a subdirectory called 1033. This is the default culture directory, as I described in "About Culture-Specific Information" in Chapter 8. (If you happen to live in a non-English-speaking country and are fortunate enough to have a

Visual Studio .NET installation for your particular language, you will likely see an additional directory here with a number representing your particular culture.)



The images directory does not have a culture directory associated with it. Therefore, whereas you will find, for example, the directory HTML\1033 containing HTML files, and the directory scripts\1033 containing the script files, you will find the images in the directory images, *not* images\1033.

In order for the different files to work together, a wizard contains a set of symbols that represent the state of the IDE user's choices while going through the wizard. With that in mind, try the following: Choose File→New→Project. When the New Project dialog box opens, on the left side choose Visual C++ Projects. On the right side, choose Win32 Project. Assign a name to the project (I called mine Win32proj1) and a directory.

When you click OK, the wizard opens. The left side of the wizard has a hyperlink (remember, this is an HTML page) for Application Settings. When you click this link, the page changes to a list of options. One of these options is Application Type, which includes a set of radio buttons, one for each application type. As is typical with radio buttons, you can select only one button at a time. Thus, you might choose a single Application Type. The HTML page also has a set of options labeled Additional Options. This set consists of Empty Project, Export Symbols, and Precompiled Header. Each of these options is a checkbox, meaning you can select multiple options; the options are not mutually exclusive. (However, the Export Symbols option is enabled only when the application type is DLL; and the Precompiled Header option is always checked, but disabled, *unless* you choose Static Library for the application type, in which case the Precompiled Header option becomes enabled.) Additionally, the page has options for ATL support and MFC support.

At this point you can either create the project or just press Cancel. Then return to the Explorer or DOS window and move down into the VCWizards\win32wiz\scripts\1033 subdirectory, found underneath the C++ installation. Inside this directory you will find the single script file that processes the information you enter into the HTML pages. This script file is called default.js. Open it in a text editor of your choice. You can see the file looks similar to a C++ or C# program in its syntax. Search on the word FindSymbol. Here are some of the lines that you will encounter:

```
var Pch = wizard.FindSymbol("PRE_COMPILED_HEADER");
wizard.FindSymbol("SUPPORT_MFC")
wizard.FindSymbol("CONSOLE_APP")
wizard.FindSymbol("WIN_APP")
```

Each of these symbols refers to an option the IDE user can choose when viewing the HTML files of the wizard. For example, the WIN_APP option refers to the user's choice of the Windows application for the Application Type.

Now move up two directories, then down to the html\1033 directory. Inside this directory you will see two .htm files. The one called default.htm is the HTML file that opens when you first start the wizard. The second, AppSettings.htm, is the page that displays the settings I described. Go ahead and open the AppSettings.htm file in a standard text editor of your choice. Now search on the string WIN_APP. Skip the first one

you come to (it refers to an image) and look at the second line, which I show you here (though spread out over several lines to accommodate the width of the book page). I've boldfaced two items that I discuss next:

```
<INPUT TYPE="radio" CLASS="Radio" onPropertyChange="InitControls();
"NAME="apptype" ID="WIN_APP" ACCESSKEY="W" TITLE="Creates a simple
Windows application. The application files include a <projectname>
.cpp file that contains the _tWinMain function, and stdafx.cpp and
stdafx.h files that are used to create the precompiled header file.">
```

You can see that the input type is radio button (the first item in bold). You can also see the button has an ID called `WIN_APP` (the second item in bold). This means the selection state of the radio button will get stored in the `WIN_APP` variable that you can access in your scripts.

Once you include an ID in the HTML control and, as a result, have a symbol defined, you can use the symbol in the *rendering* of your template files, which is a topic I cover in the next section, "Rendering the Template Files." As a preview, I'll say this: When you render a template file, you copy the file from the templates directory to the project directory, and the wizard engine automatically replaces instances of your symbol within the file with the string or value stored in the symbol. For example, if the symbol is called `GUID_COCLASS`, and the value is `543E3E5D-40DA-4AAC-8C17-1481B30B693E`, when you render a source file, the wizard engine will replace all instances within the file of `[!output GUID_COCLASS]` with `543E3E5D-40DA-4AAC-8C17-1481B30B693E`.

In order to use a symbol in your scripts, you call the wizard engine's `FindSymbol` function, as in the following code:

```
var bAttributed = wizard.FindSymbol("ATTRIBUTED");
```

To use the `FindSymbol` function, you pass the name of the symbol and save the results in a JavaScript variable.

There are three sources of symbols in your wizard:

- *HTML files.* You can define symbols in your HTML files, as I demonstrated with the previous HTML code.
- *JavaScript files.* Your JavaScript files can create symbols by calling into the wizard engine.
- *Default symbols.* Your JavaScript files can access several default symbols, such as `TEMPLATE_PATH`, which gives the location of the template files.

To create symbols in your JavaScript files, you call the wizard engine's `AddSymbol` function. Here's an example from the default.js script for the MFC Application Wizard:

```
wizard.AddSymbol("ABOUTBOX_FONT_SIZE", "8");
```

This line creates a symbol called `ABOUTBOX_FONT_SIZE` and sets it equal to the string "8".



Depending on your level of HTML expertise, it may interest you to know that you can type scripts directly into your HTML files (rather than having them in their own .js files). These scripts, when embedded in an HTML file, can call into the wizard engine just as the your default.js script file and other .js files can. However, when the script is inside an HTML file, instead of using the object name wizard, as you do in .js files, you use the object name window.external.

You can also create symbols in the scripts within your HTML files, by calling the same `AddSymbol` method; and, as usual, instead of the name `wizard`, you use the name `window.external`, as in the following line:

```
window.external.AddSymbol("HTML_VIEW", true);
```

In the previous list I also mentioned default symbols. The wizard engine provides many default symbols that your scripts can access. For these symbols' names, you do not need to call `AddSymbol`. Here are some of the more common predefined symbols:

HTML_PATH. This is the path where the HTML files reside. Normally, it will be the wizard's path, followed by the directory name `HTML`, followed by the locale (such as `1033`, which is the default).

IMAGES_PATH. This is the path where the image files reside. It will typically be the wizard's path, then the directory name `Images`.

PRODUCT_INSTALLATION_DIR. This is the root of the particular product for which the wizard works, such as `C++`. For example, the default product installation for `C++` is `c:\Program Files\Microsoft Visual Studio .NET\Vc7\`.

PROJECT_NAME. This is the name of the project, the name the IDE user typed into the New Project dialog box.

PROJECT_PATH. This is the path to the project, the project path the IDE user typed into the New Project dialog box, followed by the project name. Remember that this symbol contains both the path and filename, not just the path.

SCRIPT_PATH. This is the path to the directory containing the scripts. Normally, it will be the wizard's path, followed by the directory name `Scripts`, followed by the local, which is `1033` by default.

START_PATH. This is the wizard's path, the path of the base directory containing the `HTML`, `Scripts`, `Images`, and `Templates` directory. It will be under the main `Wizards` directory; for example, for the `MFC Application Wizard` it will be `c:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\mfccappwiz`.

TEMPLATES_PATH. This is the path where you can find the template files. Normally, it will be the wizard's path, followed by the name `Templates`, followed by the locale, which, again, is `1033` by default.

WIZARD_NAME. This is the name of the wizard, such as `mfccappwiz`.



To see the full list of predefined symbols, open the Visual Studio .NET online help. Go to the Contents and drill down to Visual Studio .NET→Visual C++→Creating and Managing Visual C++ Projects→Designing a Wizard→Files Created For Your Project→The .vsz File→Custom Parameters in the Wizard .vsz File.

You can control the values of the predefined symbols from with your .vsz file. (Remember, the .vsz file is the file in the projects directory that provides information to Visual Studio .NET about the wizard.) For example, if you add the line:

```
Param="IMAGES_PATH = c:\MyImages"
```

to your file, the IMAGES_PATH symbol will get set to "c:\MyImages". Make sure, however, that you follow the format precisely: Start with the word `Param` with no space after it, then an equal sign with no space after it, followed by a string in double quotes. The format of this string in double quotes must also be exact. Start with the name of the symbol, then a space, which is *mandatory*; next put an equal sign, followed by another mandatory space; finally, put the value for the symbol, without its own double quotes around it. (The final double quotes are for the string itself.)

You can also create your own symbols that you can pass into the scripts, as in this line from a .vsz file.

```
Param="MYSYMBOL = SOMENAME"
```

With this line in your .vsz file, the scripts will now be aware of the symbol `MYSYMBOL`, and the symbol's value will be `SOMENAME`.

When you create a wizard, you may have multiple HTML pages, and if you give your users the option to go to the various pages and make selections, they may choose to skip some of the pages. For the selections on those pages, then, you'll want to provide defaults for the symbols your HTML files set. The one HTML file that always opens when the wizard runs is called `default.htm`; therefore, put the defaults in `default.htm`. The defaults go in the `<HEAD>` section, and they look like this:

```
<HEAD>
  <SYMBOL NAME="APP_BASE_CLASS" TYPE="text" VALUE="CWinApp"></SYMBOL>
  <SYMBOL NAME="ATTRIBUTED" TYPE="checkbox" VALUE="true"></SYMBOL>
</HEAD>
```

These two `SYMBOL` lines provide default values for the `APP_BASE_CLASS` and `ATTRIBUTED` symbols. For the former, which is a text box, the default is the string "CWinApp". For the latter, which is a checkbox, the default is the value `true`. (A checkbox has only two values, `true` and `false`.)

Rendering the Template Files

When you copy the files from the templates directory to the project directory, you use a process called *rendering*. The wizard engine can render files by scanning through the

text in the files, replacing various strings with other strings, and then copying the resulting file into the project directory. Or, in the case of binary files, you can optionally copy files directly without attempting to process them as text.

As an example of this process, open the following file in the text editor of your choice:

```
C:\Program Files\Microsoft Visual Studio .NET\
Vb7\VBWizards\ComClass\Templates\1033\ComClass.vb
```

This is the template file that gets translated into the ComClass.vb file when you create a new COM class in your VB.NET project. You can see this process at work if you right-click a VB.NET project in the Solution Explorer, choose Add→Add New Item, scroll down, and choose COM Class. (Although this is for an item template rather than a project template, the concept is the same, and this ComClass.vb file is a good example.) Here's one line from the template file:

```
Public Const ClassId As String = "[!output GUID_COCLASS]"
```

This line gets translated into a line such as the following:

```
Public Const ClassId As String = "543E3E5D-40DA-4AAC-8C17-1481B30B693E"
```

However, when you create a COM class, you will see a different GUID appear on this line. In order to render the Connect.vb file, the wizard engine replaces the text [!output GUID_COCLASS] with the GUID 543E3E5D-40DA-4AAC-8C17-1481B30B693E. Where did this GUID come from? From the script. The script called a function to obtain a unique GUID and stored it in the symbol called GUID_COCLASS. Then, when the wizard engine renders the ComClass.vb file, the engine sees the symbol GUID_COCLASS, preceded by the characters [!output, and followed by the character]; the engine replaces the entire string with the value stored in the symbol. (If the script engine only sees GUID_COCLASS without the preceding and following characters, the engine will not replace the string.)

Now open the script file C:\Program Files\Microsoft Visual Studio .NET\Vb7\VBWizards\ComClass\Scripts\1033\default.js. Here are the two lines of code from the script that generate the GUID and store it in the GUID_COCLASS symbol:

```
var strRawGuid = wizard.CreateGuid();
wizard.AddSymbol("GUID_COCLASS", wizard.FormatGuid(strRawGuid, 0));
```

The first line generates a GUID and stores it in the strRawGuid variable. The second line takes the strRawGuid variable and writes it to the symbol called GUID_COCLASS, which is the symbol in the ComClass.vb template file. Thus, each time the wizard runs, a new GUID ends up in the project's ComClass.vb file.

Now take a look at the template file C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\mfccappwiz\templates\1033\childfrm.cpp. This file is part of the MFC Application Wizard, and this wizard is for a project not a project item. (To see the MFC Application Wizard, choose File→New→Project; in the New Project dialog box on the left side, choose Visual C++ Projects, and on the right side, choose MFC Application.)

The `childfrm.cpp` template file contains several symbol replacements as you saw in the `ComClass.vb` template file. However, this file also contains some if-statements. Here's one if-block:

```
[!if PROJECT_STYLE_EXPLORER]
#include "[!output TREE_VIEW_HEADER]"
#include "[!output VIEW_HEADER]"
[!endif]
```

The if lines go inside brackets; the left bracket is followed by an exclamation point and then the word `if`. This time the symbol in question, `PROJECT_STYLE_EXPLORER`, is not set by the script file but by an HTML file. The symbol is initialized by the `default.htm` file found in the directory `C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\mfccappwiz\html\1033`. Here's the line from the `default.htm` file:

```
<SYMBOL NAME="PROJECT_STYLE_EXPLORER" TYPE="radio"
VALUE="false"></SYMBOL>
```

The symbol is then set by this line from the `AppType.htm` file, found in the same directory (I've broken the line up into four lines so it fits on the page):

```
<INPUT TYPE="radio" CLASS="Radio" ACCESSKEY="x"
TITLE="Select browser-style user interface." NAME="projtype"
VALUE="radiobutton" ID="PROJECT_STYLE_EXPLORER"
onClick="OnProjectStyle();">
```

When the user of the HTML page (that is, the user of the wizard) checks or unchecks the radio button, the `PROJECT_STYLE_EXPLORER` symbol gets set to `true` or `false`, respectively. (The user unchecks the radio button implicitly by checking another radio button in the group.) When the user checks the radio button, the `OnProjectStyle` function runs. This function is found in the same `AppType.htm` file, and you're welcome to take a look at it if you want; it sets other symbols based on the project type that you select.

To compare two symbols, you can use two comparison operators that have the same syntax as their C++ equivalents: `==` and `!=`. You can also use `+` and `-` and to combine two numeric symbols. And you can embed if-statements, like so:

```
[!if PROJECT_STYLE_EXPLORER]
[!if LIST_VIEW]
```

and then, later, end the inner if-statement:

```
[!endif]
```

and later still, to end the outer if-statement:

```
[!endif]
```

You can also use AND and OR relationships using the standard C++ like `&&` for AND and `||` for OR, as in the following two lines:

```
[!if APP_TYPE_MDI && SPLITTER]

and

[!if HTML_VIEW || HTML_EDITVIEW]
```

Here's an example of a logical NOT operator, shown by an exclamation point immediately before the symbol name:

```
[!if !DOCVIEW]
```

You can use an else-block in your if-statement:

```
[!if MYSYMBOL]

and

[ !else ]

and

[!endif]
```

You also can use a limited for loop. One way is to specify an exact count, as here:

```
[!loop = 5]
...
[!endloop]
```

This loops five times.

Another way is to use a symbol that contains a numeric value:

```
[!loop = MYCOUNT]
...
[!endloop]
```

Reading a template file can be a bit confusing at times. Take a look at this line from the `childfrm.cpp` template file:

```
[!output CHILD_FRAME_CLASS]:::~~[!output CHILD_FRAME_CLASS]()
```

This generates a destructor header line as in:

```
MyChildFrm::~MyChildFrm()
```

To recap what is happening with these symbols, when you render a template file, the wizard engine replaces the symbols and processes the if-statements to generate a final file. The wizard engine then copies the final file into the project directory. Here, then, is a sample line from a script file that renders a template file:

```
wizard.RenderTemplate(strTemplate, strTarget, bCopyOnly, true);
```

The `RenderTemplate` function copies a template file (whose path and filename is given by the first parameter, `strTemplate`) to a final file (given by the second parameter, `strTarget`). The third parameter specifies whether to render the file or to just copy it without processing the text. Pass `true` if you want to copy the file without processing the text (as in the case of a binary file) or `false` if you do want to process the file. For the final parameter, pass `true` if you want to overwrite any preexisting file during the copy process. (The third and fourth parameters are optional; the defaults are both `false`, meaning the `RenderTemplate` function will process the text file and will not overwrite an existing file.)

Be careful if you choose `false` for the fourth parameter of `RenderTemplate` (or if you simply take the default), because, here, the opposite of *overwrite* is *append*. Thus, if you choose `false` for the final parameter, instead of overwriting, each time your wizard runs, the target file will be appended to the end of the existing target file. If you're working with project or source files, this is probably *not* what you want. In most cases, then, you will want to pass `true` for the final parameter.



If you look at the `default.js` file for the MFC Application Wizard, you will see it contains no calls to `RenderTemplate`. The reason is that the file instead calls the `AddFilesToProject` function, which is in a common JScript file called `common.js`. I discuss this file in more detail in “The `common.js` File” section later in this chapter.

Wizard Properties

When you interact with the wizard engine from within a script, you use the object called `wizard`, as in the following line of code:

```
var strProjectName = wizard.FindSymbol("PROJECT_NAME");
```

This line of code calls the `wizard` object's `FindSymbol` method. The `wizard` object is an instance of the `VCWizCtl` class, which implements the COM interface `IVCWizCtrlUI`. This is not the same interface that a wizard normally implements, the `IDTWizard` interface. That's because the wizard engine actually implements both interfaces, and your script interacts with the engine through the `IVCWizCtrlUI` interface by using the `VCWizCtl` object called `wizard`.

The `VCWizCtl` object has numerous members; you can see the whole list by looking up `VCWizCtl` in online help index. Here are some of the more important members that you will be using. First, two properties:

ActiveXControls. You use this object to obtain a reference to an ActiveX object.

For example, the line:

```
fso = new ActiveXObject("Scripting.FileSystemObject");
```

will store a reference to the `FileSystemObject` in the `fso` variable. You can then use the `fso` to access the file system. (To see the properties and methods for the `FileSystemObject`, look up `FileSystemObject` in the online help index.)

dte. This is a reference to the main DTE object that you can use just as you would in macros and add-ins.

Now here are a few of the many member functions.

AddSymbol. Call this function to add a symbol to the symbol namespace. Pass the symbol name as the first parameter, and the symbol value as the second parameter. For the second parameter you can pass any type.

CreateGuid. This function returns a GUID in the form of a string. The string is surrounded by curly braces and includes the hyphens in the GUID.

DoesFileExist. Pass a single string to this function. The string contains a path and filename. The function returns a Boolean value indicating whether the file exists.

FindSymbol. Call this function to retrieve a value from the symbol namespace. Pass the name of the symbol. If the symbol is not present, the function returns a 0, meaning you can call the function, like so:

```
if (!window.external.FindSymbol("MY_SYMBOL"))  
{  
    window.external.AddSymbol("MY_SYMBOL", "Hello");  
}
```

GetSystemLCID. This function returns the identifier for the current locale. By default, the identifier is 1033.

Navigate. Call this function to open up an external browser window. Pass the URL (such as `www.wiley.com`) for the first parameter, and the number 1 or 0 for the second parameter. The second parameter is an enumerator specifying whether to open the URL in an external window. The function is borrowed from the `ItemOperations` object, and that object allows you to pass 0, meaning the current window. However, in this context, the `Navigate` function always opens in an external window, so it does not matter if you pass 0 or 1.

OKCancelAlert. Call this function to display a message box to the IDE user, along with an OK button and a Cancel button. Pass a string containing the message to display. The function returns `true` if the user clicks OK and `false` if the user clicks Cancel.

RemoveSymbol. Call this function to remove a symbol from the symbol namespace. Pass the symbol name.

RenderTemplate. Call this function to render a file from the templates directory.

RenderTemplateToString. Call this function to render a file to a string. When you do so, the entire contents of the file will be returned by this function in the form of a string. Here's an example:

```
str1 = wizard.RenderTemplateToString("myfile.txt");
```

YesNoAlert. This function displays a message box to the user, along with Yes and No buttons. Pass the message as a string. The function returns `true` if the user clicks Yes and `false` if the user clicks No.

Regarding the type you can pass to `AddSymbol`, look closely at this code from a `default.js` script:

```
var x = 10;
x = x + 1;
wizard.AddSymbol("NUMBER", x);
var y = wizard.FindSymbol("NUMBER");
y = y + 1;
wizard.OKCancelAlert(y);
```

The first line declares a variable called `x` and stores the number 10 in it. The next line adds 1 to `x`. The third line adds a symbol called `NUMBER`, and stores the value of `x` in it, which is 11. The next line creates a new variable called `y`, and retrieves the value of the symbol called `NUMBER`. The next line adds 1 to the value. The final line displays the value of `y`. Since the symbols can hold any type, not just strings, this piece of code functions correctly. (If the symbols could only hold strings, it would probably throw an exception.)

A Script Wizard Tutorial

Using the information in the preceding sections, you are now ready to build a wizard script. To begin, use a text editor of your choice and create the following text file, which you will save as `testwiz1.vsz` in the `C:\Program Files\Microsoft Visual Studio .NET\Vc7\vcprojects` directory:

```
VSWIZARD 7.0
Wizard=VsWizard.VsWizardEngine
Param="WIZARD_NAME = testwiz1"
Param="WIZARD_UI = TRUE"
```

This will tell the Visual Studio .NET about your wizard, including the location. Remember, the IDE obtains the location of the wizard based on the following:

- The wizard directory for the current product.
- The name of the wizard. This comes from the `WIZARD_NAME` line in the `.vsz` file.

The IDE combines these two items to get the directory for your wizard. In this case, that will be `C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\testwiz1`.

In addition to the .vsz file, you can also have an icon file representing the icon that will be displayed in the New Project dialog box. For this example, I chose to simply copy another icon, rather than create my own. You can pick any of the .ico files in one of the project directories and copy it to the same directory as the .vsz file and call it testwiz1.ico. (Its name *must* match the name of the .vsz file, but it will have the .ico extension.) The one I copied was C:\Program Files\Microsoft Visual Studio .NET\Common7\IDE\Extensibility Projects\Visual Studio Add-in.ico. (If you create your own icon, make sure the icon is 32 by 32 pixels in size and 16 colors.)

Now you have the information the IDE needs for displaying the icon in the New Projects dialog box, and for the IDE to find the wizard. Next you need to create the wizard. The first thing you need to do is create the proper directory structure for the wizard. Here are the directories you will need to create for this example:

- C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\testwiz1
- C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\testwiz1\html
- C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\testwiz1\html\1033
- C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\testwiz1\scripts
- C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\testwiz1\scripts\1033
- C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\testwiz1\templates
- C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\testwiz1\templates\1033

These directories comprise the root directory, the HTML and its 1033 directory, the scripts and their 1033 directory, and the templates and their 1033 directory. Remember, your files will go inside the 1033 directories.



If your computer is set up for a different culture, you can change the 1033 to your own culture if you want. That way the IDE will be able to locate the files for your specific culture. However, 1033 is the default, and so if you use 1033, you will still be able to use the script. (If you want to see an example of different locales in action, take a look at C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\mfccappwiz\templates. This directory has several locales under it.)

Now comes the fun part, where you will create the files. Remember, the files will sit inside the 1033 directories. There are three such directories, and you will create one file in each directory except for the templates\1033 directory, where you will create two files. You will create a script file, which will perform the project creation. You will create an HTML file, which will provide the user interface for the wizard. You will also create .cpp and .h files in the templates\1033 directory, which will be rendered into the final project. These two template files will contain various symbols that the wizard engine will replace with strings based on the user's selections in the user interface.

Now we look at the script. This script does very little: First it grabs some directory names and saves them in variables, next it creates the project, then it renders the two template files. Type this script into a text editor and save it in the scripts\1033 subdirectory with the filename default.js.

```
function OnFinish(selProj, selObj)
{
    var temppath = wizard.FindSymbol("TEMPLATES_PATH");
    var projname = wizard.FindSymbol("PROJECT_NAME");
    var projpath = wizard.FindSymbol("PROJECT_PATH");
    var proj = CreateProject(projname, projpath);
    wizard.RenderTemplate(
        temppath + "\\class.cpp",
        projpath + "\\" + projname + ".cpp",
        false,
        true);
    wizard.RenderTemplate(
        temppath + "\\class.h",
        projpath + "\\" + projname + ".h",
        false,
        true);
    proj.Object.AddFile(projpath + "\\" + projname + ".cpp");
    proj.Object.AddFile(projpath + "\\" + projname + ".h");
}
```

First, you will notice that the name of the function in this script is `OnFinish`. That is the function that you override. The idea is that after the IDE user has filled out the GUI items, the `OnFinish` method runs, which completes the task of creating the project.

The first line of the function obtains the path of the templates directory. Notice that nowhere in the script did I need to tack on 1033 to the directory name; the wizard engine found the correct locale, meaning that my `temppath` includes the 1033 directory in the string. The second line obtains the project name, and the third line obtains the project path.

The fourth line creates the project. I passed the project name and the project path. I saved the results in a `proj` variable, which I use later. The next line (which actually spans five lines) renders the `class.cpp` template file. Notice, however, that I change the name of the template file: Instead of rendering it as `class.cpp`, I render it with the project name followed by `.cpp`. The final line (again spanning five lines) renders `class.h`, again changing its name to the project name followed by `.h`. Thus, if your project is called `MyProject`, then your two files in the project will be called `MyProject.cpp` and `MyProject.h`.

Finally, the last two lines add the two new files to the project. But just because they were rendered into the project directory doesn't mean they will be added to the project. You must *explicitly* add them, using the `AddFile` function. But notice what I'm doing here: I'm taking the `proj` variable, accessing its `Object` member, and calling the `AddFile` function. If this looks familiar, it's because the `proj` variable is an instance of `Project`, one of the extensibility classes. The `VCProject` object has an `Object` property, which is of class `VCProject`, which in turn has an `AddFile` function. As you can see, then, you can interact with the DTE objects in your scripts.

Now for the two template files. Put these files in the `templates\1033` subdirectory. Here's the first one; call it `class.cpp`:

```
#include <iostream>
#include "[!output PROJECT_NAME].h"

using namespace std;

void main() {
    [!output CLASS_NAME] inst;
    cout << inst.ToString() << endl;
}
```

You can see this is a basic C++ source file, with a couple of include lines, a using namespace line, and a main routine. However, notice the second include line has a symbol name instead of a filename. Remember, the script file earlier renders the header file's name with the project name followed by the `.h` extension. Thus, the include line needs the correct header filename, which is the project name followed by the `.h` extension. That's why I put the symbol expansion there. The same is true for the first line inside the main routine. This line creates an instance of the class name. Instead of hard-coding a class name, I put the value of the `CLASS_NAME` symbol. Inside the wizard GUI, the IDE user can type in a class name; that class name will end up in this first line within the main.

Here's the second one; call it `class.h`:

```
#include <string>
class [!output CLASS_NAME] {
public:
    [!if INCLUDE_CONST]
        [!output CLASS_NAME]() {}
        ~[!output CLASS_NAME]() {}
    [!endif]
    std::string ToString() {
        return "[!output CLASS_NAME]";
    }
};
```

Once again, this header file uses the `CLASS_NAME` symbol for the class name, instead of a hard-coded name. Further, one of the options in the wizard GUI was whether to include the constructor. Therefore, I wrap the constructor (and the destructor, too) inside an if-statement, which checks the value of the `INCLUDE_CONST` symbol. Finally, inside the `ToString` member function, I return a string constant. But that string constant is the value of the `CLASS_NAME`.

At this point, believe it or not, the wizard is ready to run, so to show how these two template files will render, go ahead and run the wizard now. You don't need to do any compilation or installation; all the files are interpreted, and you created the files in their installation location.

To launch the wizard, start up Visual Studio .NET. You can either open an existing solution or you can let the Add Project dialog box create a solution automatically for

you when you create the project. Then choose File→New→Project. When the New Project dialog box opens, in the Project Types, choose Visual C++ Projects. In the Templates, you should see your new wizard, testwiz1, along with the icon that you copied from elsewhere. From there, create a project as you normally would, by typing in a name and a location, and, if you have a solution already open, selecting either Add to Solution or Close Solution. Then click OK.

At this point, your wizard will open. It won't look like much (refer back to Figure 12.1). You will be able to type in a class name and you will be able to choose whether to generate a constructor. You can either click Cancel, to abort the creation of the project, or you can click Finish. (This GUI does not feature multiple pages in the wizard; if it did, the other pages would be separate HTML files. I show you how to do this in the sidebar titled "Using the Wizard-Wizard.")

If you click Finish, your script will begin running and you will see a new project get created and added to your current solution. Or, if you don't have a solution open, you will first see a new solution get created and your project added to it. Although this project creation feature is coded into your script, the capability to automatically create a solution if you don't have one already open is built into the IDE, so you don't need to code it into your scripts. Next, your script will render the two files and then add them to the project. When all is done, you will see your project in the Solution Explorer with the two files, just as any other project. And you will see that the filenames of the two rendered files will be the same as the project name, but with the .cpp and .h extensions.

If you double-click the .h file, it will open in the editor and you can see how the symbols were rendered. For example, when I ran the script, I called my class `GreatClass`, and I chose to add a constructor. Here's the resulting .h file:

```
#include <string>
class GreatClass {
public:
    GreatClass() {}
    ~GreatClass() {}
    std::string ToString() {
        return "GreatClass";
    }
};
```

And here's the resulting .cpp file:

```
#include <iostream>
#include "Project1.h"

using namespace std;

void main() {
    GreatClass inst;
    cout << inst.ToString() << endl;
}
```

These files can be compiled without error. As you can see, the wizard did its job properly.

The common.js File

If you look through existing default.js script files, you might notice they contain calls to various functions that are neither present in the script file nor part of the wizard object. These functions exist in a file called common.js, which contains common routines that are useful in creating the projects.

Each installed product (C++, C#, VB.NET) has its own common.js file. These files are not identical; rather, they provide routines that are more useful to the particular language. The common.js files exist in a 1033 directory off the main wizards directory for each installation. Here are the directories where you can find the common.js files:

- C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\1033
- C:\Program Files\Microsoft Visual Studio .NET\Vb7\VBWizards\1033
- C:\Program Files\Microsoft Visual Studio .NET\VC#\VC#Wizards\1033

There are too many functions in these files to list here, so I encourage you to take a look at the files and see what functions are useful to you in your script writing. (All the functions in the C++ version of common.js are documented in the online help.) However, one function in particular that I want to bring to your attention is the `AddFilesToProject` function in the C++ version of common.js. This function shows up frequently in the wizards that ship with Visual Studio .NET. If you're writing a wizard that has several files in the templates directory, it can become rather cumbersome to go through all the files, rendering them one by one, and adding them one by one, to the project, with each iteration hard-coded into your default.js script file.

Fortunately, you don't have to. The common.js file for C++ has a function called `AddFilesToProject`. To use this function, simply create a text file called `Templates.inf` and put the file in your templates\1033 directory. Inside the `Templates.inf` file, list the files that you want rendered. And, note, this file is itself a template file, meaning you can embed the symbol comparisons in it. Here's an example `Template.inf` file from the MFC DLL wizard that ships with Visual Studio .NET:

```
readme.txt
root.cpp
[!if DLL_TYPE_REGULAR || DLL_TYPE_REGULAR_STATIC]
root.h
[!endif]
root.def
[!if AUTOMATION]
root.idl
[!endif]
stdafx.h
stdafx.cpp
resource.h
all.rc
root.rc2
```

This file indicates, first, that the `readme.txt`, `root.cpp`, `root.def`, `stdafx.h`, `stdafx.cpp`, `resource.h`, `all.rc`, and `root.rc2` files will always be rendered, regardless of the settings the IDE user chooses in the wizard GUI; and, second, that the `root.h` file will render

only if either the `DLL_TYPE_REGULAR` symbol is set to `true` or the `DLL_TYPE_REGULAR_STATIC` symbol is set. Finally, the `root.idl` file will be rendered only if the `AUTOMATION` symbol is set to `true`.

For the `testwiz1` wizard that I showed you in the previous section, the `templates\1033` directory contains only two files, `class.cpp` and `class.h`, and they always get rendered regardless of the project settings. This implies a very simple `Templates.inf` file. You can, therefore, simplify your wizard a bit by creating the following text file, calling it `Templates.inf` file, and putting it in the `templates\1033` directory:

```
class.cpp
class.h
```

Next, you can modify your script, as follows:

```
function OnFinish(selProj, selObj)
{
    var temppath = wizard.FindSymbol("TEMPLATES_PATH");
    var projname = wizard.FindSymbol("PROJECT_NAME");
    var projpath = wizard.FindSymbol("PROJECT_PATH");
    var proj = CreateProject(projname, projpath);
    var InfFile = CreateInfFile();
    AddFilesToProject(proj, projname, InfFile);
}

function GetTargetName(strName, strProjectName, strResPath, strHelpPath)
{
    if (strName == "class.cpp") {
        return strProjectName + ".cpp";
    }
    if (strName == "class.h") {
        return strProjectName + ".h";
    }
    return strName;
}

function SetFileProperties(projfile, strName)
{
    return false;
}

function DoOpenFile(strTarget)
{
    return false;
}
```

As you can see, this script is longer than the previous script because it has three additional functions. However, if you're creating a wizard that generates a very lengthy script, and you have numerous files in your templates directory, then this will save significant space.

Note in this code how I simultaneously rendered the files and added them to the project: First I called `CreateInfFile`, which sets up the use of the `Templates.inf` file. Then I called `AddFilesToProject`, which does the hard work of rendering and adding each file to the project.

The three functions that follow are called by the `AddFilesToProject` function. These functions are required; without them, you will get an exception. You can use the first one, `GetTargetName`, if you want to modify one of the names. I did just that in my `GetTargetName` function: Although the template filenames are `class.cpp` and `class.h`, I wanted them to have the name of the project, followed by `.cpp` or `.h`, respectively. So in the `GetTargetName` function, I checked for the template filename and returned the modified filename.

For the `SetFileProperties` and `DoOpenFile` functions, I simply returned `false`. (Even if, as in my case, you don't want these functions to do anything, you still need to include them.) The `SetFileProperties` function takes as a parameter a `ProjectItem` object and the original template filename (not the rendered filename). You can use the `ProjectItem` object to set various configuration properties. (The return value of your `SetFileProperties` function is ignored, so if you process the item, you don't have to return `true`.) The `DoOpenFile` function, on the other hand, instructs the wizard to automatically open the file. In this function, you check the filename passed into the function; and if you want to immediately open the file into the source editor, return `true`; otherwise return `false`.



If you want to see an example where `SetFileProperties` and `DoOpenFile` don't simply return `false`, take a look at the `C:\Program Files\Microsoft Visual Studio .NET\VC#\VC#WizardsCSharpConsoleWiz\Scripts\1033\default.js` script file. This is the console wizard for a C# file, and it makes use of these functions.

DEBUGGING A SCRIPT

Debugging a script is a bit awkward at first, but once you get the hang of it, it's rather easy. To debug a script, anywhere in your script add the line:

```
debugger;
```

That line is equivalent to setting a breakpoint. Then, when you run the script, you will get a message that an exception occurred. You then will be asked if you wish to debug, and, if so, which debugger. Depending on which version of Visual Studio you have installed, you may be given the choice of several different debuggers. You can choose whichever you want and then click Yes. (I use the Visual Interdev, which shipped with Visual Studio 6.0, for the sole reason that it loads faster than Visual Studio .NET.) An instance of your chosen debugger will then start and you can trace through the code as you would any other program you are debugging.

Viewing Your HTML Files



In this section I assume you have Internet Explorer's script debugging turned off. So if you want your files to behave the same as I describe here, turn off debugging by opening up Internet Explorer. Choose Tools → Internet Options; click on the Advanced tab. Find the Disable script debugging option and uncheck it. (Checking it will turn on script debugging.)

If you look at the HTML files that exist in the various wizard directories, you will quickly see an enormous amount of information, and if you try to open the HTML files in Internet Explorer, you will find that the pages don't look like they do when the wizard runs. The reason for the difference in appearance is in one block of scripting code embedded in the HTML files:

```
<STYLE TYPE="text/css">@import url();</STYLE>
<SCRIPT>
    var strURL = window.external.FindSymbol("PRODUCT_INSTALLATION_DIR");
    strURL += "VCWizards/";
    strURL += window.external.GetHostLocale();
    strURL += "/NewStyles.css";
    document.styleSheets(0).imports(0).href = strURL;
</SCRIPT>
```

This code might vary a bit depending on who wrote the script, but the idea will be the same: The script calls into `FindSymbol` method of the `VCWizCtl` object (represented in the HTML files as `window.external`, and in the JScript files as `wizard`). When you pass `PRODUCT_INSTALLATION_DIR` to `FindSymbol`, you will get back the installation directory for the current product—*not* the installation of Visual Studio .NET, but rather C++ or VB.NET or C#. (For example, the default product installation directory for C++ is `C:\Program Files\Microsoft Visual Studio .NET\Vc7\`.) The next line adds the `VCWizards` path to the string. (Notice the script allows paths to have a forward slash.) Then the script adds the locale, which by default is 1033. (See “About Culture-Specific Information” in Chapter 8 for more information.) The script then adds the `NewStyles.css` file. Thus, by default, for C++, this script creates the following string: `C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards\1033\NewStyles.css`

This `NewStyles.css` file is an HTML style sheet, which sets the font sizes and styles, the background color, the margins, and other layout information. When you open the wizard in Visual Studio .NET, the wizard engine is able to locate the `NewStyles.css` file. The reason is that the Visual Studio .NET IDE hosts the Internet Explorer control that displays the HTML file. In doing so, the IDE assigns the `VCWizCtl` to the `window.external` object. The window object is part of the Internet Explorer object model and represents the host containing the Internet Explorer control. The host can then store an object reference in the `window.external` object. In the case of Visual Studio .NET, the IDE stores the `wizard` object in the `window.external` object.

USING THE WIZARD-WIZARD

Included with the wizards that come with Visual Studio .NET is a wizard called Custom Wizard (which I prefer to call Wizard-Wizard). This wizard creates a starting point from which you can create your own wizards. When you run this wizard, you can choose whether you want a user interface with your wizard (that is, whether you want HTML files) and, if so, how many HTML pages you want in your wizard. You also get to choose the name of your wizard. When you run the Wizard-Wizard, you will end up with a .vsz file already installed in the wizard directory for the product installation, along with a set of directories arranged in the proper hierarchy. These directories are HTML, Images, Scripts, and Templates. Additionally, you will have a directory off the root, called 1033, which contains style sheet information for the HTML files. Finally, each directory will contain starter files from which you can modify the code.

Using the techniques in this chapter, you will be well on your way to easily modifying this code and creating your own wizard. I recommend using this wizard, but with one modification: The .vsz file that the wizard generates causes the IDE to point to the installation directory for your wizard. I suggest you copy the entire installation into the main wizard directory for the product and modify the .vsz file, to give the name without the location. That way you can easily ship the wizard to others without including absolute paths in it.

Now here's the problem in displaying your HTML files *inside* Internet Explorer but *outside* the wizard: The preceding script fails because the `window.external` object is not set to a `VCWizCtl` object, causing an exception. You can see the error if you turn on Internet Explorer's script debugging. Here's the error that I see:

```
A Runtime Error has occurred.
Do you wish to debug?
Line 10
Error: Object doesn't support this property or method.
```

(Turn script debugging back off now, since it tends to get in the way when you're surfing the Web.)

Following is an updated form of the preceding script that includes an exception handler. In the exception handler I've hard-coded the path to my local style sheet file, which enables me to look at my HTML files in Internet Explorer, or even in the HTML editor in Visual Studio .NET. Here's the revised script:

```
try {
    var strURL = "../..../";
    strURL += window.external.GetHostLocale();
    strURL += "/NewStyles.css";
    document.styleSheets(0).imports(0).href = strURL;
}
catch (e) {
    var strURL =
```

```
        "C:/Program Files/Microsoft Visual Studio .NET" +  
        "/Vc7/VCWizards/1033/NewStyles.css";  
    document.styleSheets(0).imports(0).href = strURL;  
}
```

When you replace the old script with this new script, you will be able to see your HTML pages in a browser as they will appear inside the wizard. (Make sure, however, that you use your own Visual Studio .NET installation directory if you installed it somewhere other than the default.)

Moving Forward

This chapter showed you how to use the wizard engine to create your own wizards. Remember, the wizard engine is simply a general-purpose COM component that provides wizard functionality. Using this component, you can easily create your own scripts and HTML files that work together to create a wizard that maintains the same look and feel as the other wizards.

In the next chapter, "Writing .NET Add-ins for Microsoft Office," which begins Part III, I show you how you can take the same concepts on writing add-ins and use them to build add-ins for other Microsoft products.

PART

Three

**VS.NET and
Other Products**

TEAMFLY

Writing .NET Add-ins for Microsoft Office

If you've been reading this book straight through, by now you're well aware of how to create add-ins for Visual Studio .NET using the `IDTExtensibility2` interface. This interface was developed by Microsoft before Visual Studio .NET came out; it was originally created for the Office 2000 line of software. In other words, now that you know how to create add-ins for Visual Studio .NET, you can use the same techniques to create add-ins for Microsoft Office products.

Introducing Office Add-ins

It's important to remember that the Visual Studio .NET IDE is not itself really a .NET program, and its add-ins are not assemblies; they're COM objects. You could, in fact, write a Visual Studio .NET add-in using Visual Studio 6.0 without using any .NET features. But when you use C# or VB.NET or C++.NET to develop an add-in, you're developing a library that is both a COM object and an assembly. The COM portion allows the library to serve as the add-in, whereas the assembly portion allows the library to access the .NET framework. (Technically speaking, the add-in library you create using Visual Studio .NET is not actually a COM library; rather, it includes a COM type library and it works together with the `mscorlib.dll` found in the `windows\system32` directory to play the part of a COM component.)

Before delving into the topic of writing add-ins for Office products, let me briefly review the concept of an add-in. When you use Visual Studio .NET to write an add-in, you create an assembly that serves as a COM add-in to a product; this add-in in turn

uses the COM automation system to automate the target application. In the case of Visual Studio .NET, you are creating a COM add-in that accesses the DTE object, which is the automation object for Visual Studio .NET. Through this DTE object, you can access all the features of the IDE that are made available through the automation model.

The same scenario holds for writing add-ins for Office products. When you write an add-in for an Office product, you first obtain a root object (simply called the Application object, which is analogous to the DTE object). Through this root object, you can control the features made available through the automation model.



Visual Studio .NET includes a wizard for creating add-ins for Office products. When you use this wizard, you may notice that you have the option of creating the add-in for products that you don't actually have installed on your computer. Theoretically, you could create add-ins for such products; however, I do not recommend doing so. You certainly want to be able to test out your add-ins before sending them to people; therefore, if you want to develop an add-in for a particular product, make sure you first obtain the product.

Here are the products for which the wizard in Visual Studio .NET lets you develop add-ins:

- Word
- Visio
- Project
- PowerPoint
- Microsoft Outlook
- FrontPage
- Excel
- Access



Be careful if you're developing add-ins for Visio. Visio 2000, although part of the Office 2000 product line, does not support add-ins. This has been a source of a great deal of confusion among developers: If you start Visio 2000 and choose Tools⇨Options and click on the Advanced tab, you will see a checkbox labeled Enable COM add-ins. But the checkbox is *disabled*. Why? For some reason, Microsoft decided to ship Visio 2000 without COM add-in capabilities, but included this checkbox in the event it ship a patch that added COM add-in support. This never happened, so if you want to use COM add-ins in Visio, you must get Visio XP/2002 or later.

You can also use the same wizard to create add-ins for the Visual Studio .NET IDE and Macros IDE just as you can with the standard Add-in wizard.

TWO KINDS OF OFFICE ADD-INS

Before Microsoft developed the concept of a COM add-in, most Office products had software development kits (SDKs) that programmers could use to write add-ins for those products. Using these SDKs along with a C or C++ compiler, the programmer would develop a DLL that the Office product could load. (The DLL would get a different extension, such as .XLL for Excel add-ins.) But these add-ins did not make use of the COM automation model, and the technique for building each add-in different from product to product. Further, you could not share a single add-in among multiple products. Therefore, Microsoft developed COM add-ins, which make use of the COM automation model, have common interfaces and can be shared among Office applications. Today's Office applications still support the older add-ins, along with the newer COM add-ins, and the products have two separate dialog boxes for the two kinds of add-ins. Normally, the user interfaces distinguish between Add-ins (the older style add-ins) and COM Add-ins (the newer COM style). Today, Microsoft encourages us to create the newer COM add-ins. Therefore, I discuss only the COM add-ins in this book.

Writing Add-ins for Other Products

In order to write an add-in for an Office application, it's a good idea to perform two tasks that will make your life easier:

- Turn on the COM Add-ins dialog box feature of the Office application.
- Create the assemblies that reference the COM system.

In the next section I show you how to perform the first of these two tasks. In the subsequent two sections, respectively, I show you how to create an Office add-in and how to perform the second of these two tasks.

Preparing the Office Application

Although the process of writing a COM add-in for an Office product is a no secret, for some reason the dialog box for controlling the COM add-ins is not accessible by default. Instead, you have to configure the menus to show the COM Add-ins dialog box. Further, you have to do this for each Office application. Here are the steps:

1. Right-click the blank area around a toolbar in the Office application.
2. Choose Customize. The Customize dialog box will open.
3. Click the Commands tab.
4. In the Categories list, click Tools.
5. In the Commands list, find the COM Add-ins command. (On my computer it's the first in the list.) Click it and drag it to the menu bar.

I dragged my COM Add-ins command to the Tools menu and put it toward the bottom. In Microsoft Word, I put it immediately under Templates and Add-ins, as that seemed like a more-or-less logical place for it. In Microsoft Excel, I put it under Add-ins.

(Remember, the Add-ins menu item is an older kind of Add-in that I'm not covering in this book.) After you complete these steps, you can open the COM Add-ins dialog box by choosing the COM Add-ins menu item.

When you create an Office add-in, you have the choice of whether to make the add-in available to all users or only to the current user. Each option works as you might expect; however, be aware that add-ins that are installed for all users do *not* show up in the COM Add-ins dialog box. At first this might seem like a bug (and I'm sure many people have reported it as a bug), but it is, in fact, a design decision by Microsoft. The rationale is that an individual user shouldn't be able to add and remove add-ins that have been installed for all users to use. You can, of course, imagine a scenario where a systems administrator installs an add-in for all users, but at least one user doesn't want to use it and tries to remove it; he or she would be unable to do so. Thus, the add-ins for all users do not appear in the COM Add-ins dialog box.

Creating the Add-in

To create the Office add-in, start up Visual Studio .NET and create a new solution. Then choose File→Add Project→New Project. The Add New Project dialog box will open. In the Project Types treeview on the left, expand the Other Projects node; underneath it, choose Extensibility Projects. In the Templates list on the right, click Shared Add-in. Type a name and location for your project and click OK.

The Extensibility Wizard will open. The first screen is simply a splash screen; click Next. In the second screen, shown in Figure 13.1, you can choose the language you prefer: C#, VB.NET, or C++ with ATL. As with add-ins for Visual Studio .NET, I usually prefer VB.NET or C#, as these two languages are more cleanly integrated to .NET.

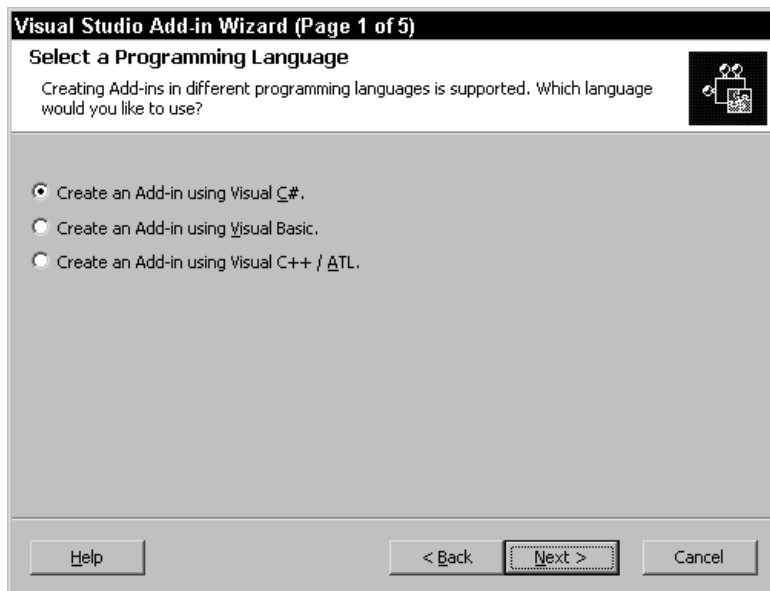


Figure 13.1 Choose the language you prefer for your Office add-in.

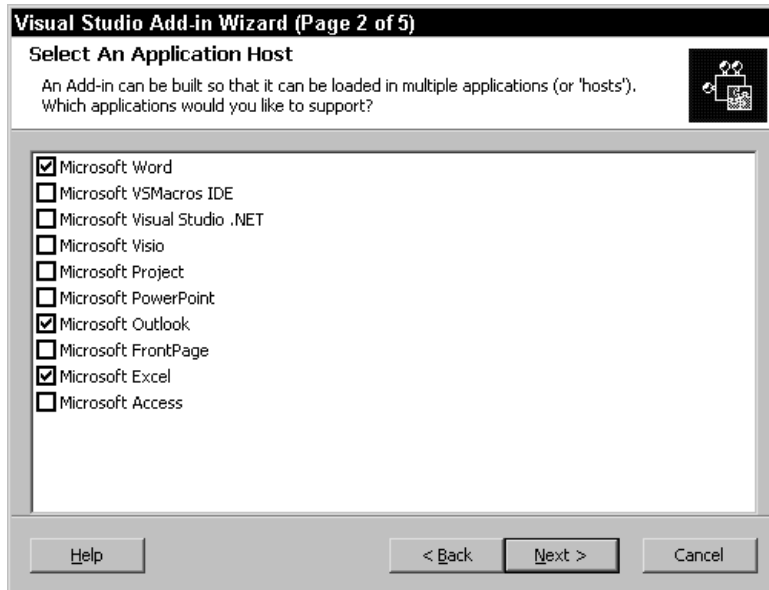


Figure 13.2 Choose the application hosts for the add-in.

After you choose your language, click Next. The Application Host screen opens, as shown in Figure 13.2. This is a list of applications you can pick for your add-in to serve. Because the interface is the same for all these products, a single add-in can support any number of these products simultaneously. (However, sometimes it's hard to think of an example of an add-in that would be useful in both—say, Microsoft Word and the Visual Studio .NET Macros IDE—although I'm sure there are examples, such as those dealing with version control and event logging.)

After you choose your application hosts, click Next. The Name and Description screen comes up next, as shown in Figure 13.3. Here you give a friendly name and a description for your add-in.

After entering a name and description, click Next. The Add-in Options screen appears, shown in Figure 13.4. Your two choices are:

- Have the add-in load when the application starts.
- Make the add-in available to all users on the computer.

When you click Next, you will see a summary of your options, shown in Figure 13.5. Look over these options, and if you want to change anything, click Back. Otherwise, click Finish to create the add-in and automatically add it to the existing solution.

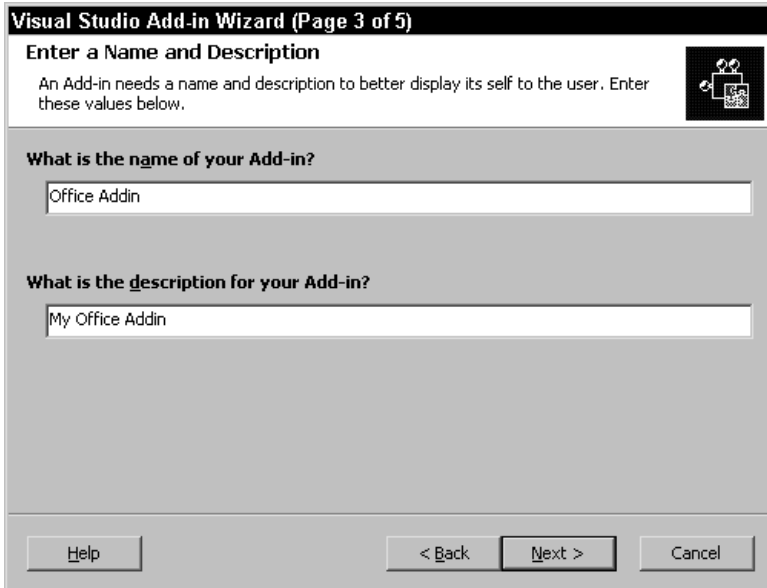


Figure 13.3 Enter a name and description for your add-in.

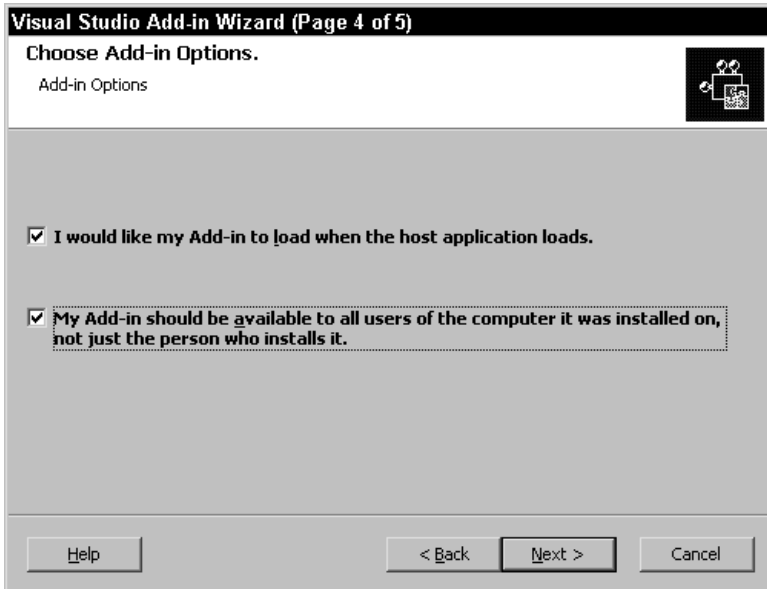


Figure 13.4 Check the options for the add-in.

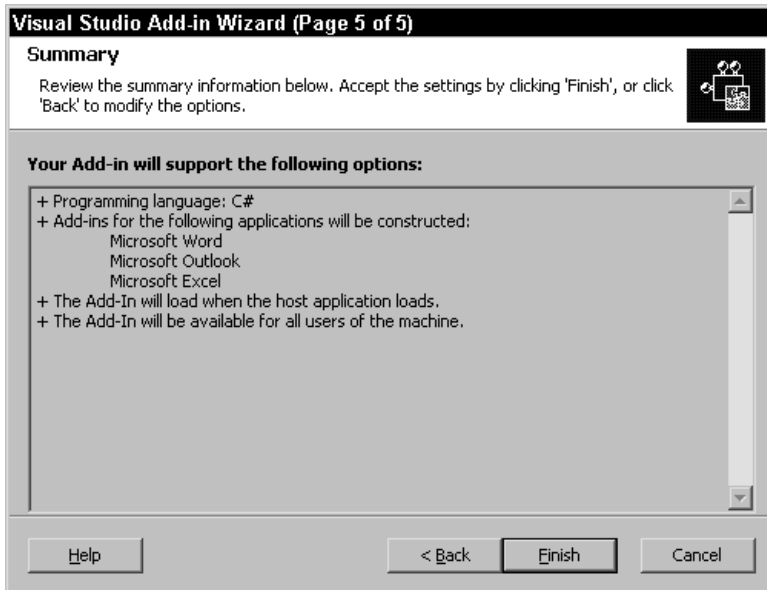


Figure 13.5 The final screen shows you a summary of your options.

Before typing in the code, you need to add references to the products you're going to support, and I show you how to do this in the next section, "Adding References to Office Products."

Adding References to Office Products

The nature of COM lets you access objects without the compiler actually knowing the object's members. If you declare an object as type `Object`, you are free to call whichever methods you want; the compiler will not issue an error. However, if those methods are not actually present when your program runs, you will get a runtime error.

When you create a COM add-in, you can use such a blind approach or you can set up a reference to a COM type library describing the objects you are accessing. Then both the IDE and the compiler will be aware of the types and can let you know if you call a method incorrectly or if you attempt to access a nonexistent member.

You have different choices for how to set up the references to the COM type libraries representing the Office products. These are:

- Run the `tlbimp` command-line tool to generate an assembly based on the COM type library.
- Add a reference to the type library file itself. This causes the IDE to run the command-line tool for you, generating the assembly for you. If you use this method, the IDE will generate the assembly every time you build your add-in.

- Download the official assemblies from Microsoft that interoperate through COM with the Office applications.
- Let the IDE generate the assembly for you, then add a reference to the generated assembly.

Of these choices, I use the third and fourth, depending on which version of Office I'm targeting. The third choice applies only to Office XP/2002. The second choice is pretty much required for Office 2000, because official assemblies are not available; and due to some issues, you cannot simply add references to some of the Office COM type libraries. (I discuss these issues in the sidebar titled "Important: Fixing Excel and Outlook.")

To get started, you simply add the references as I describe in the second option in the preceding list. After the following important note, I show you how to do this.



In this section I describe the process for adding references to the COM objects that represent the Microsoft Office products. This process will work with Office 2000 and later; however, if you're using Office XP/2002, Microsoft recommends that you instead download the prebuilt interop assemblies. These interop assemblies contain the official type information and, as such, are deemed the PIAs, which stands for Primary Interop Assemblies. To obtain the PIAs for Office XP, visit <http://support.microsoft.com> and enter 328912 into the search box to locate Knowledge Base Article 328912. (Note: As of November 2002, Microsoft's Knowledge Base articles no longer begin with a "Q." Thus, what previously would have been Q328912 is now 328912. At the time of this writing, the direct link to the article is:

<http://support.microsoft.com/default.aspx?sd=msdn&scid=kb;en-us;328912>

Moreover, at the time of this writing, the article contained a broken link to the download page. Here's the direct link:

<http://msdn.microsoft.com/downloads/sample.asp?url=/MSDN-FILES/027/001/999/msdncompositedoc.xml>

To add a reference to the Office products, start by right-clicking the References item in the Solution Explorer under your project name; and in the popup menu choose Add Reference. The Add Reference dialog box will open.

The next step depends on whether you're using the PIAs with Office XP. If you are using the PIAs with Office XP, click the PIA in the list under the .NET tab. You will have one PIA for each Office product, and the name will be, for instance, Microsoft.Office.Interop.Outlook.dll. Then click Select (or double-click the name). Do this for each Office product, then click OK. You're finished; the remainder of this section applies only if you're not using XP or the PIAs.

If you are not using Office XP (that is, you're using Office 2000), or if you are but are not using the PIAs, then you can instead reference the COM component. Remember, you will be making some changes, as I describe in the sidebar, "Important: Fixing Excel

and Outlook,” but this is what you do first: In the Add Reference dialog box, click the COM tab. A few moments might pass as the dialog box gathers up the COM components. Scroll down the list until you get to the items whose names start with “Microsoft.” Then find the products you are supporting. You might see more than one for a single product, as in:

- Microsoft Excel 5.0 Object Library
- Microsoft Excel 9.0 Object Library

Obviously, these are both for Excel, but you want the 9.0 version if you’re developing for Office 2000. Don’t use the 5.0 or any earlier version. If you’re developing for Office XP, the version will be 10.0. Click the COM component for the Office product you want to support and then click Select. Repeat this for all the products your add-in is supporting. When you’re finished, click OK.

When you do this process, the IDE takes the COM library, finds the file containing the type information (in the case of the Office products, the type files have an OLB extension and are in the C:\Program Files\Microsoft Office\Office directory), runs the `tlbimp` utility for you, and places the resulting assembly in your project’s output directory. Interestingly, the IDE does the type library importing each time you build your project as well. For this reason, when you fix these assemblies, you need to remove the reference from the COM library and add a reference directly to the fixed assembly.

Now’s a good time to read through the sidebar I’ve been referencing.

IMPORTANT: FIXING EXCEL AND OUTLOOK

If you simply import the COM libraries for the Office applications, you will find that everything is fine until you try to work with the events, at least when working with Excel and Outlook. The applications can send out event notifications (for instance, Excel can notify your add-in when the Excel user creates a new workbook). Unfortunately, for Excel and Outlook, the types for the events have a private, rather than public, access level, making them inaccessible to your add-ins. (This is a bug that Microsoft is aware of.)

The following process is clumsy at best, but it’s the official Microsoft-sanctioned way of fixing the problem, as described in its Knowledge Base articles 309336 and 316653, so please follow these steps very carefully. (Note: If you’re not working with events, then you really don’t need to do this process.)

- 1. Make sure you have already added a reference to the COM component for the Office product your add-in is supporting. (You will be *removing* this reference shortly.)**
- 2. From the Windows Start Menu, choose Programs⇨Microsoft Visual Studio .NET⇨Visual Studio .NET Tools⇨Visual Studio .NET Command Prompt.**
- 3. Inside the Command Prompt, “cd” to the output directory for your project. (That’s the bin directory, possibly followed by a configuration name such as debug.) In this directory you will find a DLL for each product you added, such as `Interop.Excel.dll`. You will be modifying this DLL.**

(continued)

IMPORTANT: FIXING EXCEL AND OUTLOOK (continued)

4. Type the following:

```
ildasm.exe /source Interop.Excel.dll /output=Interop.Excel.il
```

substituting Outlook for Excel if you're doing this for Outlook. Press Enter. This command disassembles the DLL and creates a Microsoft Intermediate Language file called Interop.Excel.il.

5. Return to Visual Studio .NET. Choose File⇨Open⇨File. In the Open File dialog box, locate the Interop.Excel.il file (or Interop.Outlook.il), click it, then OK. You are opening this file in Visual Studio .NET.
6. When the Interop.Excel.il (or Interop.Outlook.il) file opens, you need to perform a carefully crafted search and replace. This is cumbersome, so here I show you how to use regular expressions to do the trick in one step. Press Ctrl+H or choose Edit⇨Find and Replace⇨Replace. In the Replace dialog box, type the following in the Find What box:

```
private{.*}_SinkHelper
```

And in the Replace With box, type:

```
public\1_SinkHelper
```

Check the Use box in the lower left, and in the drop-down box next to this box, choose Regular expressions. Click Replace All. (This is why it's important to learn regular expressions. Without them you would have to manually change some 30 lines.)

7. Save the Interop.Excel.il (or Interop.Outlook.il) file and close it.
8. Remove the reference to Excel or Outlook (whichever you're currently working on, but only that one) by right-clicking the name Excel or Outlook in the References section under your project's name in the Solution Explorer, and in the popup menu choosing Remove.
9. Return to the Command Prompt window. In the same directory as step 4, type the following and press Enter:

```
ilasm.exe /dll Interop.Excel.il / output=Interop.Excel.dll
```

(Substitute Outlook if you're doing this for Outlook.) This rebuilds the DLL based on the modified IL code.

10. Return to Visual Studio .NET. Right-click on the References name under your project in the Solution Explorer and choose Add Reference. In the Add Reference dialog box, choose Browse. In the Select Component dialog box, switch to the Bin directory of your project, and click either Interop.Excel.dll or Interop.Outlook.dll, whichever you just updated (but only that one). Click Open. Back in the Add Reference dialog box, click OK.

That's it. The assembly is "repaired" and can now handle events properly. At this point you should have the following files in your bin directory. For Excel: Interop.Excel.dll, Interop.Office.dll, and Interop.VBIDE.dll. For Outlook: Interop.Outlook.dll. You will also have a couple .res files and .il files, which at this point are not needed. Fortunately, you only have to do this once for the project.

Writing the Add-in

Now you're ready to write the code for the add-in. Following is a simple add-in that supports Microsoft Word and Excel. Simply for variety, I chose C# for this add-in. This add-in displays a dialog box that allows you to quickly back up all the files in a source directory to a target directory, optionally putting the files in a subdirectory under the target directory named for the current date and time. (This is an add-in I plan to keep and use, as it provides an archiving feature.)

To create this add-in, create a new Shared Add-in project and follow the instructions in the preceding sections for adding references to Excel and Word. Here's the code for the Connect class:

```
namespace CSharpGeneral
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;
    using System.Runtime.InteropServices;

    [GuidAttribute("9A98275B-F196-48BD-AC0E-1DD90934A543"),
    ProgId("CSharpGeneral.Connect")]
    public class Connect : Object, Extensibility.IDTExtensibility2
    {
        public Connect()
        {
        }

        public void OnConnection(object application,
            Extensibility.ext_ConnectMode connectMode,
            object addInInst, ref System.Array custom)
        {
            applicationObject = application;
            addInInstance = addInInst;
        }

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

        public void OnAddInsUpdate(ref System.Array custom)
        {
        }

        public void OnStartupComplete(ref System.Array custom)
        {
            // Alternatively, if you know the application, you can
            // do this:
        }
    }
}
```

```
// Word.Application WordApp = applicationObject;
// CommandBars bars = WordApp.CommandBars;
// which is handy because the IDE pops up the
// members of WordApp, showing CommandBars as
// a member.

CommandBars bars = (CommandBars)applicationObject.
    GetType().InvokeMember("CommandBars",
        System.Reflection.BindingFlags.GetProperty,
        null, applicationObject, null);
CommandBar bar;
CommandBarButton button;
bar = bars["Standard"];
try
{
    // Look for an existing button
    button = (CommandBarButton)bar.Controls["Backup"];
}
catch
{
    object omissing = System.Reflection.Missing.Value ;
    button = (CommandBarButton)bar.Controls.Add(
        1, omissing, omissing, omissing, omissing);
    button.Caption = "Backup";
    button.Style = MsoButtonStyle.msoButtonCaption;
}
// Microsoft recommends that you always include a tag,
// which is a description of the button.
button.Tag = "Backup Files";
button.OnAction = "!<CSharpGeneral.Connect>";
button.Visible = true;
button.Click += new Microsoft.Office.Core.
    _CommandBarButtonEvents_ClickEventHandler(
        this.MyButton_Click);
}

public void OnBeginShutdown(ref System.Array custom)
{
}

private void MyButton_Click(CommandBarButton cmdBarButton,
    ref bool cancel)
{
    BackupForm form = new BackupForm();
    form.ShowDialog();
}

private object applicationObject;
private object addInInstance;
}
}
```

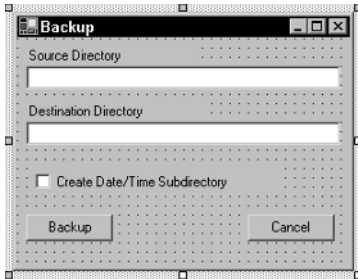


Figure 13.6 The completed form with the labels, edit controls, a checkbox, and buttons.

This add-in uses a form, but unlike Visual Studio .NET add-ins, it is a popup form, not a tool window; therefore, I do *not* use the `VSUserControlHost` that I described in “Using the Form Designer with a Tool Window” in Chapter 7, “Creating Add-ins for the IDE.” To create this form, simply right-click the project name in the Solution Explorer, and in the popup menu choose `Add→Add Windows Form`. The form that you will be designing is shown in Figure 13.6.

Now here’s the code for the form, in which you can see the various properties I set in the designer, as well as the event handlers that I added. Notice that I make use of the `System.IO` classes to perform the file backup operations. Also notice the special way I create the subdirectory name by passing a string to the `DateTime` instance’s `ToString` method.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.IO;

namespace CSharpGeneral
{
    public class BackupForm : System.Windows.Forms.Form
    {
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.Button button2;
        private System.Windows.Forms.TextBox SourceText;
        private System.Windows.Forms.TextBox DestText;
        private System.Windows.Forms.CheckBox CreateSubdirCheck;
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        public BackupForm()
        {
```

```
        InitializeComponent();
    }

protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if(components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose( disposing );
}

#region Windows Form Designer generated code
private void InitializeComponent()
{
    this.label1 = new System.Windows.Forms.Label();
    this.label2 = new System.Windows.Forms.Label();
    this.SourceText = new System.Windows.Forms.TextBox();
    this.DestText = new System.Windows.Forms.TextBox();
    this.CreateSubdirCheck = new
System.Windows.Forms.CheckBox();
    this.button1 = new System.Windows.Forms.Button();
    this.button2 = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    // label1
    //
    this.label1.Location = new System.Drawing.Point(8, 8);
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(152, 16);
    this.label1.TabIndex = 0;
    this.label1.Text = "Source Directory";
    //
    // label2
    //
    this.label2.Location = new System.Drawing.Point(8, 56);
    this.label2.Name = "label2";
    this.label2.Size = new System.Drawing.Size(160, 16);
    this.label2.TabIndex = 1;
    this.label2.Text = "Destination Directory";
    //
    // SourceText
    //
    this.SourceText.Location = new System.Drawing.Point(8, 24);
    this.SourceText.Name = "SourceText";
    this.SourceText.Size = new System.Drawing.Size(272, 20);
    this.SourceText.TabIndex = 2;
    this.SourceText.Text = "";
}
```

```
//
// DestText
//
this.DestText.Location = new System.Drawing.Point(8, 72);
this.DestText.Name = "DestText";
this.DestText.Size = new System.Drawing.Size(272, 20);
this.DestText.TabIndex = 3;
this.DestText.Text = "";
//
// CreateSubdirCheck
//
this.CreateSubdirCheck.Location = new System.Drawing.
    Point(16, 112);
this.CreateSubdirCheck.Name = "CreateSubdirCheck";
this.CreateSubdirCheck.Size = new System.Drawing.
    Size(216, 24);
this.CreateSubdirCheck.TabIndex = 4;
this.CreateSubdirCheck.Text =
    "Create Date/Time Subdirectory";
//
// button1
//
this.button1.Location = new System.Drawing.Point(8, 152);
this.button1.Name = "button1";
this.button1.TabIndex = 5;
this.button1.Text = "Backup";
this.button1.Click += new System.EventHandler(
    this.button1_Click);
//
// button2
//
this.button2.Location = new System.Drawing.Point(200, 152);
this.button2.Name = "button2";
this.button2.TabIndex = 6;
this.button2.Text = "Cancel";
this.button2.Click += new System.EventHandler(
    this.button2_Click);
//
// BackupForm
//
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(288, 197);
this.Controls.AddRange(new System.Windows.Forms.Control[] {
    this.button2,
    this.button1,
    this.CreateSubdirCheck,
    this.DestText,
    this.SourceText,
    this.label2,
    this.label1});
this.Name = "BackupForm";
this.StartPosition = System.Windows.Forms.
```

```
        FormStartPosition.CenterParent;
        this.Text = "Backup";
        this.ResumeLayout(false);

    }
    #endregion

    private void button1_Click(object sender, System.EventArgs e)
    {
        if (!Directory.Exists(SourceText.Text)) return;
        if (!Directory.Exists(DestText.Text)) return;
        String SourceDir = SourceText.Text;
        String DestDir;
        if (CreateSubdirCheck.Checked)
        {
            DateTime now = DateTime.Now;
            DestDir = Path.Combine(DestText.Text,
                DateTime.Now.ToString("yy.MM.dd.HH.mm"));
            Directory.CreateDirectory(DestDir);
        }
        else
        {
            DestDir = DestText.Text;
        }
        foreach (String filename in Directory.GetFiles(SourceDir))
        {
            String DestFile = Path.Combine(DestDir,
                Path.GetFileName(filename));
            File.Copy(filename, DestFile, false);
        }
        MessageBox.Show("Finished copying.");
        Close();
    }

    private void button2_Click(object sender, System.EventArgs e)
    {
        Close();
    }
}
}
```

After you get this add-in built and you run Word or Excel, you will be able to install the add-in by choosing **Tools** → **COM Addins**. (If you don't see this menu item, refer to "Preparing the Office Application" earlier in this chapter.) You will see your backup add-in listed, possibly along with other add-ins, depending on what you've installed. If you check the box next to the add-in, when you close the dialog box you will see a button with the word **Backup** appear on the **Standard** toolbar. If you click this, the form you created will open, allowing you to back up your file.

Debugging for Multiple Products

If you create a single add-in that will support multiple products (for example, you write a single add-in to be used in both Word and Excel), debugging the add-in can be a bit frazzling. Remember that an add-in essentially becomes a part of the target program; the example add-in, then, would be parts of both Word and Excel. Although you have only a single copy of the add-in on your hard drive, if you run Word and Excel simultaneously, you are running two separate copies of your add-in in separate address spaces. Since a debugger (such as the Visual Studio .NET IDE's debugger) can only debug a single application at a time, a single instance of Visual Studio .NET can debug either Word or Excel but not both at once. If you want to debug both at once, you have to start another instance of Visual Studio .NET and run two simultaneous debug sessions, one in each Visual Studio .NET instance. But such simultaneous debugging sessions would probably be a bit confusing. Your other option is, simply, to not debug the two target applications, Word and Excel, simultaneously.

To simplify debugging of different target applications, I recommend setting up different configurations for your project, with each configuration set to launch a different application. For example, you would create one configuration for debugging the add-in under Excel, one for debugging the add-in under Word, and so on for each application.

Remember that (for better or for worse) your configurations are tied to the solution. Here are the steps to set up a special configuration for debugging an Excel add-in:

1. Open the Configuration Manager (either right-click the solution name in the Solution Explorer, and in the popup menu choose Configuration Manager, or choose Build⇨Configuration Manager).
2. When the Configuration Manager opens, click the Active Solution Configuration drop-down list and choose New (as shown in Figure 13.7). The New Solution Configuration dialog box will open.

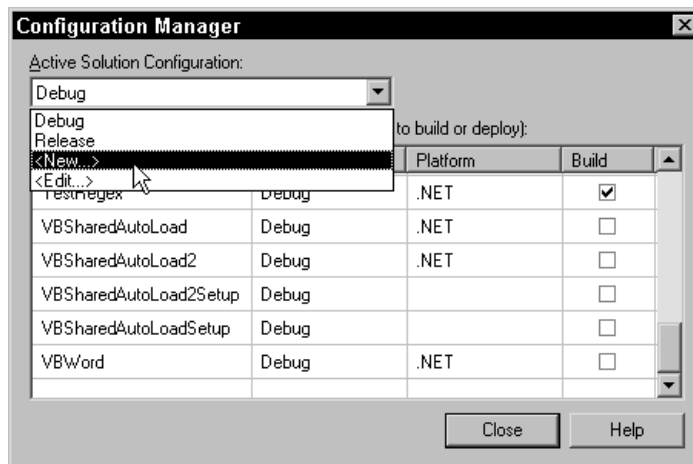


Figure 13.7 Use the Configuration Manager to add a separate Excel configuration.

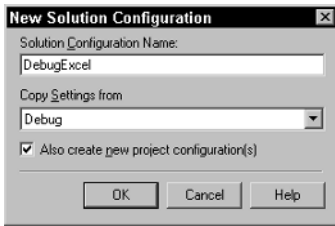


Figure 13.8 Use the New Solution Configuration to create the new configuration.

3. Type in a name for the configuration (such as DebugExcel), then choose Debug from the Copy Settings From drop-down list (as shown in Figure 13.8). Check Also Create New Project Configuration(s), if it's not checked already. Then click OK. Close the Configuration Manager.
4. Open your add-in project's Property Pages (by right-clicking the project, and in the popup menu choosing Properties). In the left treeview, choose Configuration Properties.
5. In the Configuration drop-down list, choose DebugExcel. (It might already be chosen as active.)
6. Under the Start Action category, click on the Start Application property and type in the full path to the EXCEL.EXE program, as shown in Figure 13.9. (By default, the path is C:\Program Files\Microsoft Office\Office\EXCEL.EXE.) Click OK.

Your project is now ready to debug an add-in that is running in Excel. You can follow these same steps to create a configuration for other Office products such as Word and Outlook. Then, when you want to switch quickly between configurations to debug a different Office product, you can easily choose the configuration from the drop-down list in the Standard toolbar of the main IDE, as shown in Figure 13.10.

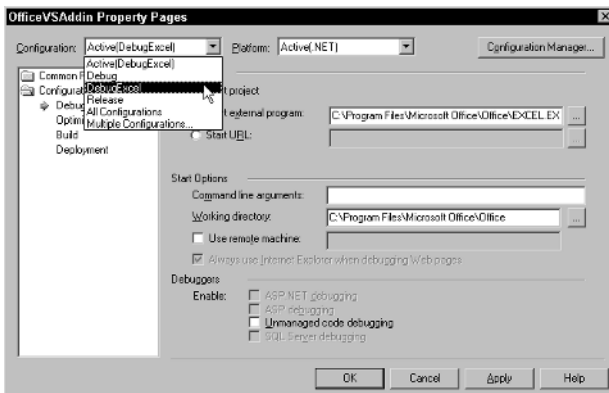


Figure 13.9 In the Property Pages dialog box, specify the path to Excel.

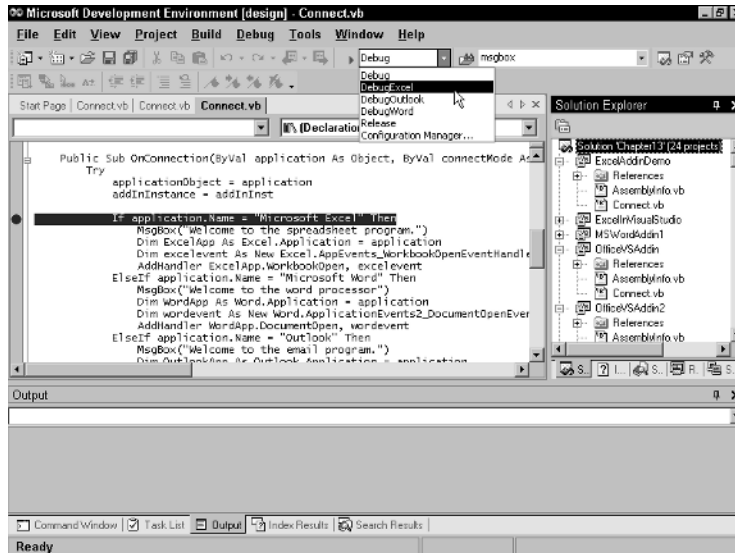


Figure 13.10 Use the Standard toolbar to quickly switch configurations.

Finally, when you are ready to debug, you can follow the same procedures as those used to debug an add-in in Visual Studio .NET. Set your breakpoints in the add-in (such as in the OnConnection function) and then run in Debug mode. But instead of Visual Studio .NET, Excel will start up.

Responding to Office Events

The Office objects include several events to which your code can respond. For example, the Excel.Application object includes an event called WorkbookOpen, which occurs when the Excel user opens a workbook or when an automation program causes Excel to open a workbook.

The application object of each Office application includes several events, as do many of the other objects within the application. For example, the Excel.Application object includes events such as NewWorkbook, SheetActivate, SheetBeforeDoubleClick, SheetBeforeRightClick, and SheetCalculate, whose names are self-explanatory. You can find out all the events by starting the Office application for which you're writing the add-in, then opening the application's Visual Basic for Applications (VBA) IDE by choosing Tools→Macro→Visual Basic Editor. Inside the VBA IDE, choose Microsoft Visual Basic Help. Inside the online help is a reference for all the objects and their events.



Before you can properly respond to events, make sure the interop assemblies are correct. Before trying out this example, refer to the sidebar, "Important: Fixing Excel and Outlook."

To respond to an event, you need to create a *delegate* object, which, in the case of a static function, is really the address of a function, or, in the case of a member function, the address of a function and an object instance. You then assign the delegate object to the event object. First I show you an example in VB.NET, followed by the same example in C#:

```
Dim excelevnt As Excel.AppEvents_WorkbookOpenEventHandler
excelevnt = New Excel. _
    AppEvents_WorkbookOpenEventHandler( _
        AddressOf ExcelWorkbookOpen)
AddHandler ExcelApp.WorkbookOpen, excelevnt
```

The first line declares the delegate type. Delegates are heavily typed, consequently you must declare the delegate specifically for the type of event it will handle. In this case, the `excelevnt` delegate will specifically handle a `WorkbookOpen` event. The second line creates the object declared in the first by calling the `New` operator, passing to the parameter the address of a function. The third line uses the VB.NET `AddHandler` keyword to assign the delegate to the `WorkbookOpen` event. Note that the `WorkbookOpen` event is a member of the `Excel.Application` object; in this code snippet, the `Excel.Application` object is called `ExcelApp`.

Now look at the same example in C#, which contains other code to make up a complete console application. I chose to demonstrate a complete application here since the main example shown later in this section is a VB.NET sample; this will enable you to see an entire C# example. Specifically, this is a simple automation program that launches Excel; it is not an add-in. Also, note that, in this example, instead of using the `WorkbookOpen` event, I demonstrate the `NewWorkbook` event, primarily because I didn't want to assume a particular workbook already existed.



When you write an add-in for an Office 2000 product, you will want to make good use of exception handlers. If an Office product encounters an error while running your add-in, and your add-in has no exception handler to catch the error, instead of notifying you as Visual Studio .NET does, the application will simply abort the add-in and set the add-in's load behavior to not run upon startup. This can be frustrating if you don't know what's happening.

```
using System;

namespace CSharpExcelAuto
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
```

```

/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main(string[] args)
{
    Excel.Application ExcelApp = new Excel.Application();
    ExcelApp.Visible = true;
    Excel.AppEvents_NewWorkbookEventHandler excelevent;
    excelevent = new
        Excel.AppEvents_NewWorkbookEventHandler(
            ExcelNewWorkbook);
    ExcelApp.NewWorkbook += excelevent;
    Excel.Workbook book = ExcelApp.Workbooks.Add(
        System.Reflection.Missing.Value );
    Excel.Worksheet sheet = (Excel.Worksheet)book.ActiveSheet;
    sheet.Cells[1,1] = "10";
    sheet.Cells[2,1] = "20";
    Excel.Range range = sheet.get_Range("A3", "A3");
    range.Formula = "=Sum(A1:A2)";

}
static void ExcelNewWorkbook(Excel.Workbook Wb)
{
    System.IO.TextWriter f =
        System.IO.File.AppendText("c:\\automation.log");
    f.WriteLine("Excel: NewWorkbook " + Wb.FullName);
    f.Close();
}
}
}

```

To use this sample, create a new C# console application and type the code shown into the Class1.cs file; then compile it. If you run the code in a command (DOS) window, you will see Excel start up, a new sheet gets created and filled with some values. Meanwhile, you can check that the c:\automation.log file was indeed created, meaning the event fired.

Next is an entire add-in that can be used in three different Office applications: Word, Excel, and Outlook. This add-in is written in VB.NET and includes the earlier code snippet demonstrating how to set up an event. To use this code, create a new Shared Add-in, choose VB.NET for the language, and choose Word, Excel, and Outlook for the supported applications. Add references to all three applications, as I described earlier in this chapter in the section “Adding References to Office Products.” Then fix up the Outlook and Excel interop assemblies.



Once you’ve fixed up the interop assemblies, you can save them in a common directory (such as C:\Program Files\Microsoft Visual Studio .NET\Common7\IDE\PublicAssemblies) and reuse them in future projects. But if you do this, don’t forget to include them when you deploy your add-in.

This add-in sets up separate event handlers based on which application is running, as you can see in the `OnConnection` function. You can also see in the `OnDisconnection` function that I unhooked the event handlers by calling the `RemoveHandler` function, which mirrors the `AddHandler` function. Unhooking the event handlers is important because if you don't, the COM system will see that the event objects are still in use, hence will not allow the Office application to exit even after the application user chooses `File` → `Exit` in the application.



When writing an add-in, always remember to unhook your events. If you find that the application for which you're writing the add-in appears to quit (its window will disappear) but the application still appears in the Task Manager's processes list, the likely solution is to add code to unhook the events. Meanwhile, to end the process, you can right-click the application name in the processes list of the Task Manager and choose End Process.

Here, now, is the code for the add-in:

```
Imports Microsoft.Office.Core
imports Extensibility
imports System.Runtime.InteropServices

<GuidAttribute("E2E96CEA-0629-4177-AEC8-34302224502E"), _
ProgIdAttribute("OfficeVSAddin.Connect")> _
Public Class Connect

    Implements Extensibility.IDTExtensibility2

    Dim applicationObject As Object
    Dim addInInstance As Object

    ' Event handlers
    Dim excelevent As Excel.AppEvents_WorkbookOpenEventHandler
    Dim wordevent As Word.ApplicationEvents2_DocumentOpenEventHandler
    Dim outlookevent As Outlook.ApplicationEvents_NewMailEventHandler

    Public Sub OnBeginShutdown(ByRef custom As System.Array) Implements _
        Extensibility.IDTExtensibility2.OnBeginShutdown
    End Sub

    Public Sub OnAddInsUpdate(ByRef custom As System.Array) Implements _
        Extensibility.IDTExtensibility2.OnAddInsUpdate
    End Sub

    Public Sub OnStartupComplete(ByRef custom As System.Array) _
        Implements Extensibility.IDTExtensibility2.OnStartupComplete
        Dim f As System.IO.TextWriter = System.IO.File.AppendText( _
            "c:\multiaddin.log")
        f.WriteLine("OnStartupComplete")
    End Sub
End Class
```

```

        f.Close()
    End Sub

    Public Sub OnDisconnection(ByVal RemoveMode As Extensibility. _
        ext_DisconnectMode, ByRef custom As System.Array) Implements _
        Extensibility.IDTExtensibility2.OnDisconnection
        ' Unhook the handlers. This is important
        ' especially for Outlook; otherwise it
        ' will hang around in memory.

    Try
        If applicationObject.Name = "Microsoft Excel" Then
            Dim ExcelApp As Excel.Application = applicationObject

            ' See text about fixing Excel interop before
            ' attempting to use this sample code!
            RemoveHandler ExcelApp.WorkbookOpen, excelevent
        ElseIf applicationObject.Name = "Microsoft Word" Then
            Dim WordApp As Word.Application = applicationObject
            RemoveHandler WordApp.DocumentOpen, wordevent
        ElseIf applicationObject.Name = "Outlook" Then
            Dim OutlookApp As Outlook.Application =
applicationObject
            RemoveHandler OutlookApp.NewMail, outlookevent
        End If

    Catch e As System.Exception
        Dim f2 As System.IO.TextWriter = _
            System.IO.File.AppendText("c:\addinerror.log")
        f2.WriteLine("OfficeVSAddin: OnDisconnection " & e.Message)
        f2.Close()
    End Try
End Sub

    Public Sub OnConnection(ByVal application As Object, ByVal _
        connectMode As Extensibility.ext_ConnectMode, ByVal _
        addInInst As Object, ByRef custom As System.Array) Implements _
        Extensibility.IDTExtensibility2.OnConnection
    Try
        applicationObject = application
        addInInstance = addInInst

    If application.Name = "Microsoft Excel" Then
        Dim ExcelApp As Excel.Application = application

        ' See text about fixing Excel interop before
        ' attempting to use this sample code!
        excelevent = New Excel. _
            AppEvents_WorkbookOpenEventHandler( _
                AddressOf ExcelWorkbookOpen)
        AddHandler ExcelApp.WorkbookOpen, excelevent
    ElseIf application.Name = "Microsoft Word" Then

```

```
Dim WordApp As Word.Application = application
wordevent = New Word. _
    ApplicationEvents2_DocumentOpenEventHandler( _
        AddressOf WordDocumentOpen)
AddHandler WordApp.DocumentOpen, wordevent
ElseIf application.Name = "Outlook" Then
Dim OutlookApp As Outlook.Application = application
outlookevent = New Outlook. _
    ApplicationEvents_NewMailEventHandler( _
        AddressOf OutlookNewMail)
AddHandler OutlookApp.NewMail, outlookevent
End If

Dim f As System.IO.TextWriter = System.IO.File. _
    AppendText("c:\multiaddin.log")
f.WriteLine("OnConnection: " & application.Name)
f.Close()
Catch e As System.Exception
Dim f2 As System.IO.TextWriter = _
    System.IO.File.AppendText("c:\addinerror.log")
f2.WriteLine("OfficeVSAddin" & e.Message)
f2.Close()
End Try
End Sub

Sub OutlookNewMail()
Try
Dim f As System.IO.TextWriter = _
    System.IO.File.AppendText("c:\multiaddin.log")
f.WriteLine("New Mail " & Now().ToShortDateString & " " _
    & Now().ToShortTimeString)
f.Close()
Catch e As System.Exception
Dim f2 As System.IO.TextWriter = _
    System.IO.File.AppendText("c:\addinerror.log")
f2.WriteLine("OfficeVSAddin" & e.Message)
f2.Close()
End Try
End Sub

Sub ExcelWorkbookOpen(ByVal Wb As Excel.Workbook)
Dim f As System.IO.TextWriter = _
    System.IO.File.AppendText("c:\multiaddin.log")
f.WriteLine("Excel: Open workbook " & Wb.FullName)
f.Close()
End Sub

Sub WordDocumentOpen(ByVal Doc As Word.Document)
Dim f As System.IO.TextWriter = _
    System.IO.File.AppendText("c:\multiaddin.log")
f.WriteLine("Word: Open document " & Doc.FullName)
f.Close()
```

```
End Sub
```

```
End Class
```

Next is a sample multiaddin.log file from running the add-in in Word, Outlook, and Excel:

```
OnConnection: Microsoft Excel
OnStartupComplete
Excel: Open workbook C:\Book2.xls
OnConnection: Microsoft Word
OnStartupComplete
Word: Open document C:\Write\MacrosAddins\Text\ch13.doc
OnConnection: Outlook
OnStartupComplete
New Mail 11/24/2002 5:32 PM
```

You can see I started up Excel, opened Book2.xls, and then started up Word and loaded the ch13.doc document (the chapter you're reading); then I started Outlook and received one new email message.

As you can see from the output, I'm using Word to write this book while developing an add-in for Word, which required me to engage in some interesting electronic acrobatics. The point is, just as you need to consider the effects of a Visual Studio .NET add-in that you're writing while using Visual Studio .NET, you also need to consider what happens if you're using various Office products while trying to develop them. In my case, any time I wanted to rebuild the add-in, I had to first shut down Word, since Word had a lock on the file, preventing me from rebuilding the add-in. (I could have investigated unloading the add-in from within Word, but I wanted to be sure everything went smoothly. Therefore, I simply exited out of Word whenever I rebuilt the add-in.)

I also encountered another interesting situation that you might be able to learn from. While researching this chapter, I had, at any given moment, five or six test add-ins running. Some of these I simply forgot about. But just because I forgot about them didn't mean they weren't still there. Consequently, when the preceding code listing wrote to a file called c:\multiaddin.log, so did *another*, earlier add-in that I was testing. That add-in, however, had an unhandled exception that occurred after opening the c:\multiaddin.log file but before closing it. This resulted in the operating system placing a lock on the file, which prevented the add-in I was working on from writing to the file, resulting in seemingly inexplicable exceptions (but at least these exceptions were handled). The moral here is to monitor the state of your add-ins to make sure you don't have any old skeleton add-ins hanging around in the closet, as I did.

To help you in this matter, here's a macro that lists all the registered add-ins. Like so many of the other macros in this book, this macro requires the VBMacroUtilities assembly, which allows the macro to write to its own Output pane.

```
Function BreakdownLoadBits(ByVal Bits As Integer) As String
    BreakdownLoadBits = ""
    ' Bit value 4 is reserved.
    ' The parens are important in these comparisons
```



```
    If (Bits And 1) = 1 Then
        BreakdownLoadBits = "Connected "
    End If
    If (Bits And 2) = 2 Then
        BreakdownLoadBits += "BootLoad "
    End If
    If (Bits And 8) = 8 Then
        BreakdownLoadBits += "DemandLoad "
    End If
    If (Bits And 16) = 16 Then
        BreakdownLoadBits += "ConnectFirstTime"
    End If
    BreakdownLoadBits += "(" & Bits & ")"
End Function

Sub ListSpecificOffice(ByVal OfficeName As String, _
    ByVal RootKey As Microsoft.Win32.RegistryKey)
    VBMacroUtilities.Print(OfficeName & " " & RootKey.Name)
    Dim regkey As Microsoft.Win32.RegistryKey
    regkey = RootKey.OpenSubKey( _
        "Software\Microsoft\Office\" & OfficeName & "\Addins")
    Dim strs As String()
    strs = regkey.GetSubKeyNames()
    Dim key As String
    For Each key In strs
        VBMacroUtilities.Print("    " & key)
        Dim subkey As Microsoft.Win32.RegistryKey
        subkey = regkey.OpenSubKey(key)
        Dim loadbehavior As Integer = subkey.GetValue("LoadBehavior")
        VBMacroUtilities.Print("        LoadBehavior: " & _
            & BreakdownLoadBits(loadbehavior))
    Next
End Sub

' The actual macro
Sub ListOfficeAddins()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    ListSpecificOffice("Outlook", Microsoft.Win32.Registry.CurrentUser)
    ListSpecificOffice("Outlook", Microsoft.Win32.Registry.LocalMachine)
    ListSpecificOffice("Word", Microsoft.Win32.Registry.CurrentUser)
    ListSpecificOffice("Word", Microsoft.Win32.Registry.CurrentUser)
    ListSpecificOffice("Excel", Microsoft.Win32.Registry.CurrentUser)
    ListSpecificOffice("Excel", Microsoft.Win32.Registry.LocalMachine)
End Sub
```

To save space, I only listed the three Office applications that I focus on in this chapter, Outlook, Word, and Excel. You're welcome to add more lines to the ListOfficeAddins subroutine for other Office applications you might have on your computer.

Moving Forward

This chapter guided you through the process of writing add-ins for various Microsoft Office products. I showed you how to add the COM Addins menu item to the Office products and how to fix Excel and Outlook if you're using Office 2000. (Remember, for Office XP, be sure to obtain the PIAs.) I also pointed out that if you're using Visio 2000, you cannot create add-ins, even though a disabled checkbox in the options dialog box would suggest you can.

In the next chapter, I take you in the opposite direction: Instead of using Visual Studio .NET to write an add-in for an Office application, I show how you can take an Office application and either embed it inside Visual Studio .NET or simply make use of the Office application's services through the COM and automation interfaces.

Integrating Visual Studio .NET with Microsoft Office

In this chapter, I show you how to integrate Visual Studio .NET with various Microsoft Office programs. This chapter is meant to be more “hands-on” than the others, therefore you’ll find more code samples.

Many products that you use on Windows—including all Microsoft Office products, as well as many products for which you can’t write add-ins—are COM servers. By this I mean that a program running on Windows (such as Visual Studio .NET) can communicate with the other programs, manipulate them, or use their services to manipulate data.

To access the other programs, you create a COM object representing the program. That object then has methods and properties *just like any other object*. In the next section I show you different ways to create a COM object for another program. Then I show you how you can determine which services the object has available; finally, I show you several examples of automating and accessing the services of other programs, such as Microsoft Word and Microsoft Outlook.

In Chapter 13, “Writing .NET Add-ins for Microsoft Office,” in the section “Adding References to Office Products,” I showed you ways to add references to other COM objects, including the Office products. Recall that once you add a reference to an Office product, you can automate the Office product from within your add-ins and macros just as you can automate Visual Studio .NET through the `DTE` object. In the following sections I use these techniques to automate the Office products, resulting in the integration of the Office products and Visual Studio .NET.

FINDING REFERENCE INFORMATION

As you develop software (including macros and add-ins) that integrates with the various Microsoft Office components, you'll need access to online help for the objects and classes. But if you look in the online help that comes with Visual Studio .NET, you won't find what you need. Instead, where you need to look is in the Office Developer Documentation, which you can find in two places: If you have the Microsoft Developer Network (MSDN) CD, the Office Developer Documentation item is immediately off the root in the table of contents; and if you don't have the MSDN CD, you can find the information online: go to <http://msdn.microsoft.com/library/default.asp>; in the tree on the left, you will see an item called Office Solutions Developer, which contains all the office documentation.

Adding a Spelling Checker

In the following example, I create an add-in that uses the spell check features of Microsoft Word to check the spelling of highlighted text in your code. Of course, such a feature is mainly useful for the comments in your code, as checking the spelling of your source code is a hopeless exercise.



This add-in uses the VSUserControlHost that I described in "Using the Form Designer with a Tool Window" in Chapter 7, "Creating Add-ins for the IDE." Refer to that section for more information on building the VSUserControlHost library. Also, remember to add a COM reference to the VSUserControlHost 1.0 Type Library when you build the following program. And be sure to add a reference to Microsoft Word, as I described in Chapter 13, "Writing .NET Add-ins for Microsoft Office."

To create the spell check add-in, add a new Visual Studio .NET add-in project (not a shared add-in project). Call the add-in SpellingAddinThis add-in uses VB.NET for the language. Like other add-ins that create a tool window, select the Tool menu option so the Connect class will implement the IDTCommandTarget interface.



When I wrote this add-in, I was in for a bit of surprise. I originally tried the line:

```
doc.Close()
```

to close the document. But I got an error:

```
C:\Write2001\VS.NET-IDE\dev\Chapter13\VBWord\Module1.vb(37):
'Close' is ambiguous across the inherited interfaces
'Word._Document' and 'Word.DocumentEvents_Event'.
```

To fix this error, I had to cast the doc object to Word._Document. You will see this cast in the code for the form.

In the following add-in, I make reference to a Document object. This is not the same Document object that you find in the DTE library for Visual Studio .NET. Rather, it is part of the Microsoft Word library. Be careful not to confuse the two.

Here's the code for the Connect module:

```
Imports Microsoft.Office.Core
imports Extensibility
imports System.Runtime.InteropServices
Imports EnvDTE

<GuidAttribute("AC9C3239-E46A-453A-9BCE-30AB383E75CB"), _
ProgIdAttribute("SpellingAddin.Connect")> _
Public Class Connect

    Implements Extensibility.IDTExtensibility2
    Implements IDTCommandTarget

    ' Note: I changed applicationObject to Public Shared!
    Public Shared applicationObject As EnvDTE.DTE
    Dim addInInstance As EnvDTE.AddIn
    Private doc As VSUserControlHostLib.IVSUserControlHostCtl = Nothing
    Private toolwin As Window = Nothing
    Public Shared wapp As Word.Application

    Public Sub OnBeginShutdown(ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnBeginShutdown
    End Sub

    Public Sub OnAddInsUpdate(ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnAddInsUpdate
    End Sub

    Public Sub OnStartupComplete(ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnStartupComplete
    End Sub

    Public Sub OnDisconnection(ByVal RemoveMode As _
    Extensibility.ext_DisconnectMode, ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnDisconnection
        CType(wapp, Word._Application).Quit(False)
    End Sub

    Public Sub OnConnection(ByVal application As Object, ByVal _
    connectMode As Extensibility.ext_ConnectMode, _
    ByVal addInInst As Object, ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnConnection

        ' Create the instance of Microsoft Word
        wapp = New Word.Application()
        wapp.Visible = False
    End Sub
End Class
```

```
applicationObject = CType(application, EnvDTE.DTE)
addInInstance = CType(addInInst, EnvDTE.AddIn)
Dim tempdoc As Object

Dim newguid As String = "{6961A5CB-51EB-4cbf-ABD3-E39671FED89A}"
toolwin = applicationObject.Windows.CreateToolWindow( _
    addInInstance, _
    "VSUserControlHost.VSUserControlHostCtl", _
    "Spelling", newguid, tempdoc)
toolwin.Visible = True
doc = CType(tempdoc, VSUserControlHostLib.IVSUserControlHostCtl)
Dim asm As System.Reflection.Assembly
asm = System.Reflection.Assembly.GetExecutingAssembly()
doc.HostUserControl(asm.Location, _
    "SpellingAddin.SpellForm")
Try
    Dim commands As Commands = applicationObject.Commands
    Dim command1 As Command = commands.AddNamedCommand( _
        addInInstance, _
        "Show", "Check Spelling", "Shows the Spelling Window", _
        True, 59, Nothing, _
        vsCommandStatus.vsCommandStatusSupported + _
        vsCommandStatus.vsCommandStatusEnabled)
    Dim viewMenu As CommandBarPopup = _
        applicationObject.CommandBars("MenuBar"). _
        Controls("&View")
    Dim viewMenuBar As CommandBar = viewMenu.CommandBar
    Dim othersMenu As CommandBarPopup = _
        viewMenu.Controls("Other Windows")
    Dim othersBar As CommandBar = othersMenu.CommandBar
    command1.AddControl(othersBar, 1)

Catch
End Try
End Sub

Public Sub Exec(ByVal cmdName As String, ByVal executeOption As _
    vsCommandExecOption, ByRef varIn As Object, ByRef varOut As Object,
    ByRef handled As Boolean) Implements IDTCommandTarget.Exec
    handled = False
    If (executeOption = vsCommandExecOption. _
        vsCommandExecOptionDoDefault) Then
        If cmdName = "SpellingAddin.Connect.Show" Then
            toolwin.Visible = True
            handled = True
            Exit Sub
        End If
    End If
End If
```

```

End Sub

Public Sub QueryStatus(ByVal cmdName As String, ByVal neededText _
As vsCommandStatusTextWanted, ByRef statusOption As _
vsCommandStatus, ByRef commandText As Object) _
Implements IDTCommandTarget.QueryStatus
    If neededText = EnvDTE.vsCommandStatusTextWanted. _
vsCommandStatusTextWantedNone Then
        If cmdName = "SpellingAddin.Connect.Show" Then
            statusOption = CType(vsCommandStatus. _
vsCommandStatusEnabled + vsCommandStatus. _
vsCommandStatusSupported, vsCommandStatus)
        Else
            statusOption = vsCommandStatus. _
vsCommandStatusUnsupported
        End If
    End If
End Sub
End Class

```

Now I will show you the form and code that you'll create for the SpellForm form. Figure 14.1 shows you the layout for the form. Remember, to create this form, right-click the project name in the Solution Explorer, and in the popup menu choose Add➤Add User Control.

```

Public Class SpellForm
    Inherits System.Windows.Forms.UserControl

    #Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call
    End Sub

```

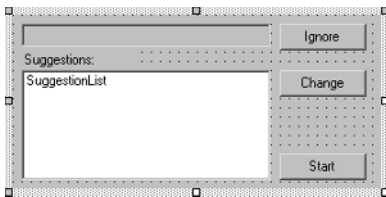


Figure 14.1 The SpellForm.vb form layout.


```
End Sub

'UserControl overrides dispose to clean up the component list.
Protected Overloads Overrides Sub Dispose(ByVal disposing As
Boolean)
    If disposing Then
        If Not (components Is Nothing) Then
            components.Dispose()
        End If
    End If
    MyBase.Dispose(disposing)
End Sub

'Required by the Windows Form Designer
Private components As System.ComponentModel.IContainer

Friend WithEvents Label1 As System.Windows.Forms.Label
Friend WithEvents IgnoreButton As System.Windows.Forms.Button
Friend WithEvents ChangeButton As System.Windows.Forms.Button
Friend WithEvents StartButton As System.Windows.Forms.Button
Friend WithEvents SuggestionList As System.Windows.Forms.ListBox
Friend WithEvents WordEdit As System.Windows.Forms.TextBox
<System.Diagnostics.DebuggerStepThrough()> Private Sub _
InitializeComponent()
    Me.WordEdit = New System.Windows.Forms.TextBox()
    Me.Label1 = New System.Windows.Forms.Label()
    Me.SuggestionList = New System.Windows.Forms.ListBox()
    Me.IgnoreButton = New System.Windows.Forms.Button()
    Me.ChangeButton = New System.Windows.Forms.Button()
    Me.StartButton = New System.Windows.Forms.Button()
    Me.SuspendLayout()
    '
    'WordEdit
    '
    Me.WordEdit.Location = New System.Drawing.Point(8, 8)
    Me.WordEdit.Name = "WordEdit"
    Me.WordEdit.ReadOnly = True
    Me.WordEdit.Size = New System.Drawing.Size(216, 20)
    Me.WordEdit.TabIndex = 0
    Me.WordEdit.Text = ""
    '
    'Label1
    '
    Me.Label1.Location = New System.Drawing.Point(8, 32)
    Me.Label1.Name = "Label1"
    Me.Label1.Size = New System.Drawing.Size(100, 16)
    Me.Label1.TabIndex = 1
    Me.Label1.Text = "Suggestions:"
    '
    'SuggestionList
```

```

    ,
    Me.SuggestionList.Location = New System.Drawing.Point(8, 48)
    Me.SuggestionList.Name = "SuggestionList"
    Me.SuggestionList.Size = New System.Drawing.Size(216, 95)
    Me.SuggestionList.TabIndex = 2
    ,
    'IgnoreButton
    ,
    Me.IgnoreButton.Enabled = False
    Me.IgnoreButton.Location = New System.Drawing.Point(232, 8)
    Me.IgnoreButton.Name = "IgnoreButton"
    Me.IgnoreButton.TabIndex = 3
    Me.IgnoreButton.Text = "Ignore"
    ,
    'ChangeButton
    ,
    Me.ChangeButton.Enabled = False
    Me.ChangeButton.Location = New System.Drawing.Point(232, 48)
    Me.ChangeButton.Name = "ChangeButton"
    Me.ChangeButton.TabIndex = 4
    Me.ChangeButton.Text = "Change"
    ,
    'StartButton
    ,
    Me.StartButton.Location = New System.Drawing.Point(232, 120)
    Me.StartButton.Name = "StartButton"
    Me.StartButton.TabIndex = 5
    Me.StartButton.Text = "Start"
    ,
    'SpellForm
    ,
    Me.Controls.AddRange(New System.Windows.Forms.Control() _
    {Me.StartButton, Me.ChangeButton, Me.IgnoreButton, _
    Me.SuggestionList, Me.Label1, Me.WordEdit})
    Me.Name = "SpellForm"
    Me.Size = New System.Drawing.Size(320, 150)
    Me.ResumeLayout(False)

```

End Sub

#End Region

```

' Remember to add reference to Microsoft Word
Private Running As Boolean = False
Private doc As Word.Document
Private WordNum As Integer = 0

Private range As Word.Range = Nothing

Private Sub SetupSpell()

```

```
IgnoreButton.Enabled = True
ChangeButton.Enabled = True
StartButton.Text = "Stop"

doc = Connect.wapp.Documents.Add()

Dim clp As System.Windows.Forms.Clipboard
clp.SetDataObject(Connect.applicationObject.ActiveDocument. _
    Selection.Text)
doc.Content.Paste()

Running = True
WordNum = 1
DoNextSpell()
End Sub

Private Sub DoNextSpell()
    range = doc.Words.Item(WordNum)
    While range.SpellingErrors.Count = 0 And _
        WordNum < doc.Words.Count
        WordNum += 1
        range = doc.Words.Item(WordNum)
    End While

    If range.SpellingErrors.Count > 0 Then
        WordEdit.Text = range.Text
        Dim spell As Word.SpellingSuggestion
        Dim spells As Word.SpellingSuggestions
        spells = range.GetSpellingSuggestions()
        SuggestionList.Items.Clear()
        For Each spell In spells
            SuggestionList.Items.Add(spell.Name)
        Next
    End If
    WordNum += 1
    If WordNum > doc.Words.Count Then
        WordEdit.Text = ""
        SuggestionList.Items.Clear()
        SuggestionList.Items.Add("No more misspellings.")
        StopSpell()
    End If
End Sub

Private Sub StopSpell()
    IgnoreButton.Enabled = False
    ChangeButton.Enabled = False
    StartButton.Text = "Start"
    Running = False
    If MsgBox("Apply changes?", MsgBoxStyle.YesNo) = _
        MsgBoxResult.Yes Then
```

```

        Connect.applicationObject.ActiveDocument.Selection.text = _
            doc.Content.Text
    End If
    CType(doc, Word._Document).Close(False)
End Sub

Private Sub StartButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles StartButton.Click
    If Running = False Then
        SetupSpell()
    Else
        StopSpell()
    End If
End Sub

Private Sub IgnoreButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles IgnoreButton.Click
    DoNextSpell()
End Sub

Private Sub ChangeButton_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles ChangeButton.Click
    Dim ind As Integer = SuggestionList.SelectedIndex
    If ind = -1 Then
        Exit Sub
    End If
    Dim replace As String = SuggestionList.Items.Item(ind)
    If Not range Is Nothing Then
        Dim trimmed As String = range.Text.TrimEnd()
        Dim pad As String = range.Text.Substring(trimmed.Length())
        range.Text = replace & pad
    End If
    DoNextSpell()
End Sub
End Class

```



Loading another program such as Microsoft Word can take some time, so I recommend that you load the program once, when your add-in begins (that is, in the `OnConnection` function), then shut down the program when your add-in unloads (in the `OnDisconnection` function). When I wrote the spell check add-in, I initially loaded Microsoft Word each time I was ready to check the spelling. But this took awhile, so I moved the load process to the `OnConnection` and then had to load Word only once. Also, don't forget to unload the program, because it won't automatically do so and your IDE users could end up with numerous instances running every time they restart your add-in.

HINT FOR PROGRAMMING OFFICE OBJECTS

Here's a hint that will make your Office programming adventures easier: If you're not sure how to accomplish something, remember that normally you do not simply create a new instance of an Office object; rather, you start with another object (such as the `Application` object) and request a new instance. For example, I wanted to write a macro that would use Outlook to email my code file to someone. To do so, I needed a new `MailItem` object. Although traditional programming experience would suggest that I simply needed to call the new operator on the `MailItem` class, I knew I would not be able to do so, thanks to my knowledge of Office programming. Instead, I looked at the `Application` object for Outlook, where I found a function called `CreateItem`. This function takes an enumerated parameter, which can be:

```
OlItemType.olAppointmentItem
OlItemType.olContactItem
OlItemType.olDistributionListItem
OlItemType.olJournalItem
OlItemType.olMailItem
OlItemType.olNoteItem
OlItemType.olPostItem
OlItemType.olTaskItem.
```

The one I wanted, of course, was `OlItemType.olMailItem`.

After you create the add-in, you can use it by starting up a new session of Visual Studio .NET and then loading the add-in as you would any other add-in. When you do, a new tool window will appear that looks just like the one you designed (refer back to Figure 14.1). (Note that, initially, the window might be kind of small; you can resize it or simply dock it.) Next highlight some comments in your code. (You can highlight code, too, but be prepared for lots of spelling errors since the Word dictionary doesn't recognize many code identifiers. That's why I recommend checking only your comments.) Then click the Start button. If the add-in finds any spelling errors, it will show them in the Spelling and Suggestions boxes, allowing you to click Ignore or Change. When the spell check session is finished, a message box will appear asking if you'd like to apply the changes. (This is slightly different from spell check sessions in Microsoft Word, which applies each change as you make it.)

Integrating an Excel Spreadsheet

If you use Excel a lot, you can easily write an add-in that lets you work with a particular Excel spreadsheet in a tool window inside Visual Studio .NET. But before I show you the add-in, I want to point out that there's a quick and easy way to load a spreadsheet into Visual Studio .NET. But I also want to point out that this method is a bit limited. To use the quickie method, all you have to do is type the full path and filename of the .xls workbook file into the URL edit box of the built-in browser. When you do so, the built-in browser will launch Excel and embed it into the browser window, displaying

the spreadsheet. However, you won't see the Excel toolbars, limiting what you can do with the spreadsheet. In contrast, the add-in I show you here does display the toolbars, adding to your capabilities.

To create this add-in, start a new Visual Studio .NET add-in project (not a shared add-in project). Call the project `ExcelInVisualStudio`, and choose VB.NET for the language of this add-in. Choose *only* Microsoft Visual Studio .NET for the host. Select the Tool menu option so that the add-in's `Connect` class will implement the `IDTCommandTarget` interface.

Next add a reference to Excel. Remember, if you're using XP and plan to support XP, use the Primary Interop Assemblies; or, if you're planning on using and supporting Excel 2000, use the Microsoft Excel 9.0 Type Library, found under the COM tab of the Add References dialog box.



This add-in does not make use of Excel's events; therefore, you can get away with not fixing it up as I described in the sidebar titled "Important: Fixing Excel and Outlook," in Chapter 13, "Writing .NET Add-ins for Microsoft Office." However, if later you plan to enhance this add-in to support events, then I recommend that you fix the Excel interop.

The goal here is to get an Excel window into a tool window. Earlier versions of Excel included a COM component (called an OCX component, which is today known as an ActiveX component) that allowed you to drop an instance of Excel into any program that could host ActiveX controls. Unfortunately, Microsoft no longer provides such a control. But Microsoft did enable an alternate way of getting Excel to appear inside another application (which is documented in its online Knowledge Base article number 304662). Internet Explorer serves two sides in the COM world: It can host COM servers such as Excel, and it can serve as an ActiveX component. So that's what I use here: I am going to place an Internet Explorer control in a tool window and then request that IE load an Excel spreadsheet, which will embed Excel inside the IE window, which is inside the tool window.

Since IE already has an ActiveX control, I don't need to use the `VSUserControlHost`, as I did in many other add-ins in this book. Instead, I can embed IE directly into a tool window. Here's the code for the `Connect` module, which does so:

```
Imports Microsoft.Office.Core
imports Extensibility
imports System.Runtime.InteropServices
Imports EnvDTE

<GuidAttribute("4B8A3716-3DA1-47CB-83BF-EB57BB61ACAE"), _
ProgIdAttribute("ExcelInVisualStudio2.Connect")> _
Public Class Connect

    Implements Extensibility.IDTExtensibility2
    Implements IDTCommandTarget
```

```
Dim applicationObject As EnvDTE.DTE
Dim addInInstance As EnvDTE.AddIn
Dim doc As SHDocVw.WebBrowser = Nothing
Dim toolwin As Window = Nothing

Public Sub OnBeginShutdown(ByRef custom As System.Array) Implements _
    Extensibility.IDTExtensibility2.OnBeginShutdown
End Sub

Public Sub OnAddInsUpdate(ByRef custom As System.Array) Implements _
    Extensibility.IDTExtensibility2.OnAddInsUpdate
End Sub

Public Sub OnStartupComplete(ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnStartupComplete
End Sub

Public Sub OnDisconnection(ByVal RemoveMode As Extensibility. _
    ext_DisconnectMode, ByRef custom As System.Array) Implements _
    Extensibility.IDTExtensibility2.OnDisconnection
End Sub

Public Sub OnConnection(ByVal application As Object, _
    ByVal connectMode As Extensibility.ext_ConnectMode, _
    ByVal addInInst As Object, ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnConnection

    applicationObject = CType(application, EnvDTE.DTE)
    addInInstance = CType(addInInst, EnvDTE.AddIn)
    Dim objAddIn As AddIn = CType(addInInst, AddIn)
    Dim CommandObj As Command
    Try
        Dim newguid As String = _
            "{5010CCC1-9C64-46d1-8460-73A92681552A}"
        Dim tempdoc As Object = Nothing
        Dim macroexp As Window = applicationObject.Windows.Item( _
            Constants.vsWindowKindMacroExplorer)
        Dim objkind As String = macroexp.ObjectKind

        toolwin = applicationObject.Windows.CreateToolWindow( _
            addInInstance, "Shell.Explorer", _
            "New Tool Window", newguid, tempdoc)
        toolwin.Visible = True
        doc = CType(tempdoc, SHDocVw.IWebBrowser)
        Dim emptyobj As Object = Nothing
        AddHandler doc.NavigateComplete2, AddressOf _
            Me.NavigateComplete2
        doc.Navigate2("file:///c:\book1.xls")
    Catch e As System.Exception
```

```

        MsgBox(e.Message)
    End Try
End Sub

Private Sub ToggleToolbars()
    doc.ExecWB(SHDocVw.OLECMDID.OLECMDID_HIDETOOLBARS, _
        SHDocVw.OLECMDEXECOPT.OLECMDEXECOPT_DONTPROMPTUSER)
End Sub

Public Sub NavigateComplete2(ByVal sender As Object, _
    ByRef URL As Object)
    Try
        Dim exceldoc As Excel.Workbook = doc.Document
        If exceldoc Is Nothing Then
            Exit Sub
        End If
        If TypeName(exceldoc) <> "Workbook" Then
            Exit Sub
        End If
        exceldoc.Windows.Item(1).Zoom = 80.0
        ToggleToolbars()
    Catch e As Exception
        MsgBox(e.Message)
    End Try
End Sub

Public Sub Exec(ByVal cmdName As String, ByVal executeOption As _
    vsCommandExecOption, ByRef varIn As Object, ByRef varOut As Object, _
    ByRef handled As Boolean) Implements IDTCommandTarget.Exec
    handled = False
    If (executeOption = vsCommandExecOption. _
        vsCommandExecOptionDoDefault) Then
        If cmdName = _
            "ExcelInVisualStudio2.Connect.ExcelInVisualStudio2" Then
            handled = True
            Exit Sub
        End If
    End If
End Sub

Public Sub QueryStatus(ByVal cmdName As String, ByVal neededText _
    As vsCommandStatusTextWanted, ByRef statusOption As _
    vsCommandStatus, ByRef commandText As Object) Implements _
    IDTCommandTarget.QueryStatus
    If neededText = EnvDTE.vsCommandStatusTextWanted. _
        vsCommandStatusTextWantedNone Then
        If cmdName = _
            "ExcelInVisualStudio2.Connect.ExcelInVisualStudio2" Then
            statusOption = CType(vsCommandStatus. _

```



```

        vsCommandStatusEnabled + _
        vsCommandStatus.vsCommandStatusSupported, vsCommandStatus)
    Else
        statusOption =
vsCommandStatus.vsCommandStatusUnsupported
    End If
End If
End Sub
End Class

```

You will notice in this code that I hard-coded the name of a spreadsheet, `c:\book1.xls`. My assumption here is that you would be working on a single spreadsheet. If that's not true for you, at the end of this section I offer some tips for enhancing this add-in. Note, however, that as it stands, you will need to create a spreadsheet whose filename is `c:\book1.xls` before running this add-in, or change the code to point to a spreadsheet file of your choice.

When you run this add-in, you will get a tool window that contains the spreadsheet. Like any tool window, you will be able to dock or close it. Like my other tool window add-ins, this one includes a menu item on the `View` → `Other Windows` menu bar that lets you reopen the tool window.

Now here are the promised tips for enhancing this add-in:

- Presently, this add-in does not include a `Show` command that displays the tool window; rather, the tool window opens when the add-in loads. You could add a command such as `Show`, and perhaps a command that includes an `.xls` filename as a parameter. The command would open the `.xls` spreadsheet file in the tool window. Using the `Command` window, the IDE user could type the command followed by a spreadsheet filename, thereby opening the spreadsheet.
- You might add a menu item to the Visual Studio .NET IDE that displays the `OpenFile` dialog box, from which the IDE user could choose a file to display. To do this, you would probably want to have the menu item execute another command in your add-in that first displays the `OpenFile` dialog box and then opens the spreadsheet. This would be a separate command from the one in the preceding paragraph.
- Since this add-in uses Internet Explorer, it can display any type of file that Internet Explorer can display, including any Office application. This means this add-in is not limited to Excel files. Simply choosing a different file type will cause your built-in Internet Explorer control to launch a different Office application.

Automating from Macros

Though you are free to automate other programs from the Visual Studio .NET macros, doing so requires climbing a short ladder to make the automation happen. Whereas a VB.NET or C# project in the main IDE can reference COM components, the macros can

reference only assemblies. Therefore, to access a COM component from a macro, you need to create an assembly wrapper. Fortunately, this is pretty easy to do. When you are working in the main IDE and you add a reference to a COM component, behind the scenes the main IDE runs the `tlbimp` program to build an assembly wrapper for the COM component. For the Macros IDE, you need to do this yourself. Here's how:

1. Open the Visual Studio Command Prompt window.
2. Use the `cd` command to switch to the directory containing the type library. In the case of Office applications such as Word, Excel, and Outlook, this is, by default, `C:\Program Files\Microsoft Office\Office`. The type library files will have various extensions, including `.tlb`, `.olb`, or even `.exe` and `.dll`. For the Office applications, the extensions are `.olb`, as in the following command output, which shows a few of the possible Office applications for Office 2000.

```
Directory of C:\Program Files\Microsoft Office\Office
```

```
03/19/1999  02:00p                638,976  EXCEL9.OLB
03/19/1999  01:31p                131,072  GRAPH9.OLB
02/01/1999  04:15p                 20,480  MSBDR9.OLB
03/02/1999  12:53p                163,840  MSOUTL9.OLB
03/17/1999  03:42p                548,864  MSWORD9.OLB
01/06/1999  05:50p                228,864  XL5EN32.OLB
           6 File(s)          1,732,096 bytes
           0 Dir(s)          238,598,656 bytes free
```

```
C:\Program Files\Microsoft Office\Office>
```

3. Run the following command, changing the application name to whichever application you're working with; this command will generate an assembly for you.

```
C:\Program Files\Microsoft Office\Office>tlbimp MSOUTL9.OLB
Microsoft (R) .NET Framework Type Library to Assembly Converter
1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```
Type library imported to Outlook.dll
```

```
C:\Program Files\Microsoft Office\Office>copy
```

4. Note the resulting assembly (in this case, `Outlook.dll`) and copy or move the assembly to the public assemblies directory. (I prefer to move the assembly, so that I don't have custom files sitting in the Office directory.) Note that the following two lines actually comprise a single line that had to be divided to fit on the page:

```
move Outlook.dll "c:\Program Files\Microsoft Visual
Studio .NET\Common7\IDE\PublicAssemblies"
```



You might be wondering why I'm doing steps significantly differently here from those outlined in Chapter 13. The reason is twofold: First, for macros to find these assemblies they must be in the publicassemblies directory, not a project directory. Second, I'm not using any events, so I didn't see a need to fix up the assemblies. But if you prefer, you can combine things here: You can fix the assemblies, and then you'll have them in the publicassemblies folder, in which case you can simply reference these fixed-up assemblies in the future, rather than fixing them up on a per-project basis.

You now have an assembly for the Outlook application or whichever application you chose. You can access all the types within this assembly just as you would any other assembly. I did this process for both Outlook 2000 (starting with the MSOUTL9.OLB file) and Excel 2000 (starting with the EXCEL9.OLB file). Here, then, are several macros that demonstrate how you can use these assemblies. These macros use my VBMacroUtilities assembly, which I described in Chapter 3, "Introducing the Visual Studio Macros IDE."



Now that you have assemblies representing the types in the COM libraries, and you access the Auto List Members option (found in the Options dialog box through Tools→Options under the All Languages section), you will see the different types and members available in the Office applications as you type the names into the editor, just as you do any other types.

Here's a quick demo of the Contacts database in Outlook:

```
Sub DemoOutlookContacts()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim outapp As New Outlook.Application()
    Dim myitems As Outlook.Items
    myitems = outapp.GetNamespace _
        ("MAPI").GetDefaultFolder _
        (Outlook.OlDefaultFolders.olFolderContacts).Items
    Dim obj As Outlook.ContactItem = _
        myitems.Find("[Company] = \"Whitehouse\"")
    While Not obj Is Nothing
        VBMacroUtilities.Print(obj.FullName & " " & obj.EmailAddress)
        obj = myitems.FindNext()
    End While
    CType(outapp, Outlook._Application).Quit()
End Sub
```

This looks up any contacts where the company is listed as Whitehouse. To make this work, I added email addresses for the U.S. president and vice president to my Contacts list (so I can make my global concerns about computers and software known). Here's a sample output:

```
Mr. President president@whitehouse.gov
Mr. Vice President vicepresident@whitehouse.gov
```

Next I show you a slightly more sophisticated macro, one that looks up any *flagged* mail items in the Outlook Inbox. (To flag an item in Outlook, double-click the email to open it in its own window; then in the toolbar, click the little flag icon.)

In this macro I use the Outlook Find feature to locate all the email messages that match a particular criteria, which here is: [FlagStatus] = 'Flagged'. (I had to do a bit of digging to figure out which words to put here. Ultimately, I realized that the best way to determine these names (such as FlagStatus and Flagged) was to open Outlook, choose Tools⇒Advanced Find, click the Advanced field, click the Field drop-down, and choose All Mail Fields⇒Flag Status. The Value drop-down list contained the possible values, including Flagged. (I used my common sense to figure that for Flag Status I had to remove the space to get FlagStatus.)

```
Sub ListFlaggedInbox()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim outapp As New Outlook.Application()
    Dim myitems As Outlook.Items
    myitems = outapp.GetNamespace _
        ("MAPI").GetDefaultFolder _
        (Outlook.OlDefaultFolders.olFolderInbox).Items
    Dim mymailitem As Outlook.MailItem = _
        myitems.Find("[FlagStatus] = 'Flagged'")
    While Not mymailitem Is Nothing

        ' Write out the subject
        ' (We can also grab the body for parsing!!!)
        VBMacroUtilities.Print(mymailitem.Subject)

        ' Reset the flag to complete. (Remember to call save!)
        mymailitem.FlagStatus = Outlook.OlFlagStatus.olFlagComplete
        mymailitem.Save()

        ' Get the next item!
        mymailitem = myitems.FindNext()
    End While
    CType(outapp, Outlook._Application).Quit()
End Sub
```

In case you're curious why, in the final line, I cast outapp to Outlook._Application, I did so because Outlook has multiple interfaces, and more than one has a Quit function. I want the one that goes with the Application interface, so I cast directly to that interface. But because I'm using an interface, not a class, I preceded the name Application with an underscore—not because that's standard but because that's the way Outlook is coded.

Now here's an interesting macro that actually puts Outlook to work. This macro takes the current source code file and emails it to someone. (I've hard-coded my Yahoo! email address, and although you're welcome to use this address, you might want to change it to one of your own to confirm that it works.)

```
Sub MailCode()
    VBMacroUtilities.Setup(DTE)
    VBMacroUtilities.Clear()
    Dim outapp As New Outlook.Application()
    Dim mitem As Outlook.MailItem
    mitem = outapp.CreateItem(Outlook.OlItemType.olMailItem)
    mitem.To = "jeffcogs@yahoo.com"
    mitem.Subject = "Hello from Visual Studio"
    mitem.Body = "This is a message from Visual Studio!"
    Dim filename As String
    filename = DTE.ActiveDocument.FullName
    Dim sendname As String
    sendname = System.IO.Path.GetFileName(filename)
    sendname = "Latest - " & sendname
    mitem.Attachments.Add(filename, _
        Outlook.OlAttachmentType.olByValue, 1, sendname)
    CType(mitem, Outlook._MailItem).Send()
End Sub
```



When you attach a file to a mail item, remove the path name of the attachment, as I did in my examples. That way, the recipient won't see an entire path name in the attachment. Also note that you can change the filename associated with the item using the fourth parameter of the `Attachments.Add` method. If you do so, however, I recommend that you maintain the filename extension (such as `.cpp` or `.vb`). If you leave off an extension, neither the recipient nor the recipient's computer will know the file type, in which case, the recipient might be afraid to open the file, the computer might not be able to open it, or the mail program might trash the mail altogether, as a virus protection feature.

To conclude this chapter, look at this rather lengthy macro I wrote that takes special fields in your source code and uses the names in these fields to extract information from an Excel spreadsheet. This can be useful if you want various data in your comments or even in your source code strings.

To try out this macro, add some comment lines to a source code file, such as:

```
' --MY_DATE-- 0 --
' --MY_TIME-- 0 --
```

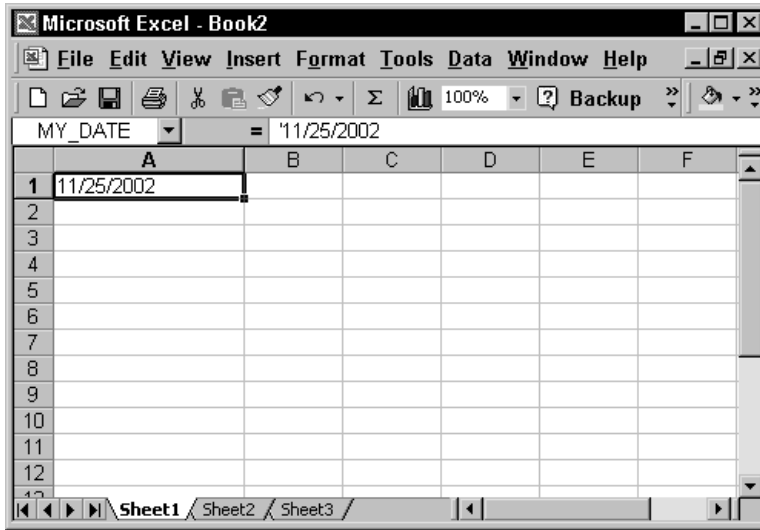


Figure 14.2 You can name a single cell and reference it through a Visual Studio macro.

Then create a spreadsheet file in Excel. (This macro has the name `c:\Book1.xls` hardcoded; if you prefer to use a different name, be sure to change the macro). In the spreadsheet file, type a formula or value into a cell and then *name* the cell the same as that in your comment (such as `MY_DATE` or `MY_TIME`, without the hyphens; the hyphens enable the macro to locate the names). To name a cell in Excel, take a look at Figure 14.2: Simply click on the cell and then, in the upper-left corner of Excel, where you see the location name of the cell (such as `A1`), type a new name and press Enter. You can see how I did this in Figure 14.2, where I named the cell `MY_DATE`. When you run this macro, the macro will replace the 0 in your comment with the value from the spreadsheet, such as:

```
-MY_DATE- 12/20/02 -
```

```
Sub UpdateFieldsFromExcel()
    ' Remember to create the Microsoft Excel
    ' assembly wrapper and reference it
    '     tlbimp excel9.olb
    ' Note: Importing Excel results in two
    ' assemblies, excel.dll and vbide.dll. You
    ' Need to put both in the publicassemblies
    ' directory.
    '
    ' Example of fields (which are named cells in the spreadsheet)
    '     --MY_DATE-- 12/20/02 --
    '     --MY_TIME-- 08:45:00 --
    Dim doc As Document = DTE.ActiveDocument
    Dim UndoWasOpen As Boolean = DTE.UndoContext.IsOpen
```

```
If Not UndoWasOpen Then
    DTE.UndoContext.Open("FillFields", False)
End If
Dim excelapp As New Excel.Application()
Dim book As Excel.Workbook = _
    excelapp.Workbooks.Open("c:\Book1.xls")
excelapp.Visible = True
Dim sheet As Excel.Worksheet = book.ActiveSheet
Dim range As Excel.Range
Dim textdoc As TextDocument = doc.Object
Dim sel As TextSelection = doc.Selection
Dim spt As EditPoint = textdoc.StartPoint.CreateEditPoint
Dim ept As EditPoint = textdoc.EndPoint.CreateEditPoint
Dim text As String = spt.GetLines(1, ept.Line + 1)
Dim re As New System.Text.RegularExpressions.Regex("--.*?--.*?--")
Dim fieldre As New _
    System.Text.RegularExpressions.Regex("--(.*)--(.*)--")
Dim ms As System.Text.RegularExpressions.MatchCollection
Dim ma As System.Text.RegularExpressions.Match
ms = re.Matches(text)
Dim itemnum As Integer
For itemnum = ms.Count - 1 To 0 Step -1
    ma = ms.Item(itemnum)
    Dim m As System.Text.RegularExpressions.Match
    m = fieldre.Match(ma.Value)
    Dim group As System.Text.RegularExpressions.Group
    If m.Groups.Count = 3 Then
        ' This regex returns the whole item as the first
        ' group member (making a total of 3), so I skip
        ' that one and just use the 2nd and 3rd (index 1 and 2).
        Dim group1 As System.Text.RegularExpressions.Group = _
            m.Groups.Item(1)
        Dim group2 As System.Text.RegularExpressions.Group = _
            m.Groups.Item(2)
        Try
            range = sheet.Range(group1.Value)
            Dim replace As String = "--" & group1.Value & _
                "-- " & range.Value & " --"
            text = text.Remove(ma.Index, ma.Length)
            text = text.Insert(ma.Index, replace)
        Catch
        End Try
    End If
Next
spt.Delete(ept)
spt.Insert(text)
If Not UndoWasOpen Then
    DTE.UndoContext.Close()
End If
End Sub
```

Notice that this macro makes heavy use of regular expressions. The regular expression `-. *?-. *?-` finds each occurrence of two hyphens followed by any characters, then two hyphens followed by any characters, ending with two hyphens. Thus, the expression will find `-MY_DATE- 12/20/02 -`. (The `. *` is shorthand for any number of any characters, and the question mark is shorthand for “do not include the hyphen characters” that follow in the regular expression.)

Moving Forward

In this chapter I showed you how you can use Visual Studio .NET to work together with various Office products. This included making use of services from Office (the example was a spell checker that used Microsoft Word’s spell check features), embedding an Office document in a tool window, and automating an Office product from a macro.

In the next chapter I continue with the theme of product integration by demonstrating how to automate Visual Studio .NET from other products, including various scripting languages. Since these scripts automate Visual Studio .NET in the same manner as the macros, in one sense you can actually write macros in these other scripting languages.

TEAMFLY

Integrating with Other Products

In this chapter I show you how you can use other languages to automate the Visual Studio .NET IDE. Any language that has access to the COM system can automate Visual Studio .NET. In a sense, this means you can write macros in any language you want, although you won't be able to use the Macros IDE to manipulate these macros.

As a self-professed computer language junkie, I have found numerous languages that work well as automation programs. In this chapter I describe two of my favorite languages, Delphi and Python, as well as the Windows Script Host and its two default languages, VBScript and JScript. You can also obtain versions of Perl and Tcl that have access to the COM system, allowing you to write macros in those languages as well. Take a look at the concepts I present in this chapter, and if you have a preferred language, give it a shot.

Windows Script Host

Starting with Windows 2000, Microsoft has included as a standard feature in all versions of Windows a Windows Scripting Host, or WSH for short. The WSH is a big secret in Windows, because for some reason very few programmers even know of its existence. If you're old enough to remember the old DOS batch files, WSH is the modern version of batch files. Or, if you're a Unix head, as many of us are, WSH scripts are akin to Unix scripts. You can use scripts to automate many processes in Windows, and in your scripts you can open applications, send keystrokes to the applications, and so on.



As an example of WSH scripts in one of my previous books, I wrote a script that automatically did a screen capture, then launched a graphics program, pasted the screen capture into the program, saved the file, and closed the program. (I also mentioned this script in Chapter 1, “All about Macros and Add-ins.”) This script was written in VBScript, which is a simplified version of Visual Basic developed specifically for scripting.

By default, the WSH language handles two languages (although technically speaking, it can support any language with the right software): VBScript and JScript. JScript is a language that has its roots in Java, although, in fact, it bears very little resemblance to Java.

The WSH scripting languages let you obtain COM objects just as you can in other languages on Windows. Because of this single feature, you can write automation programs that control the Visual Studio .NET through the DTE object.

To begin this discussion, look at this VBScript program that, first, obtains the DTE object, then obtains the list of Document objects. It then steps through that list, using a `for each` construct, obtaining the name of each document, appending them to a string, and tacking on a carriage return after each string. Finally, after obtaining the list of documents, the script displays a message box showing the names of the documents.

Type this script into a text editor and save it with a `.vbs` filename extension (I called mine `runvsnets.vbs`).

```
Dim WshShell
set WshShell = WScript.CreateObject("WScript.Shell")

Function Control()
    Dim dte, docs, doc
    set dte = GetObject(, "VisualStudio.DTE")
    set docs = dte.Documents
    text = ""
    name = ""
    for each doc in docs
        name = doc.FullName
        text = text + name
        text = text + chr(13)
    next
    WshShell.Popup(text)
End Function

Control()
```

There are two ways you can run the script. By default, files with a `.vbs` extension are associated with the script engine. Thus, if you locate your `.vbs` file in a folder, you can double-click the file and Windows will run it in the script engine. Or you can manually start up the script engine. One way is through the DOS command prompt. If you're in the directory containing your script file, you can type the following:

```
wscript runvsnet.vbs
```

But a little-known fact about DOS prompts on WinNT-based systems, including Windows 2000 and XP, is that you can run a program from the DOS prompt simply by typing in the filename. If you have a file called, for example Letter.doc, and you type:

```
Letter.doc
```

the DOS shell will launch Microsoft Word, opening Letter.doc. Thus, you can simply run your script by typing its filename at the prompt, provided you're in the directory containing the file:

```
runvsnet.vbs
```

You can also do this trick from the Run box, which is accessible from Start Menu⇨Run. In the Run dialog box, if you type the full path to the file (such as c:\scripts\runvsnet.vbs), or if you browse to the file, you can directly run the program.



If you're new to the Windows Script Host engine, I encourage you to explore it. WSH is both powerful and surprisingly useful for automating repetitive tasks in Windows. For information about WSH, go to <http://search.microsoft.com> and type WSH into the search form.

If, however, you prefer a more C++ like syntax, you can use JScript instead of Visual Basic. Here's the same script as the preceding one, rewritten in JScript. The code is pretty much a one-to-one translation, with a syntax that resembles C++. Save this file with a .js extension, which by default is associated with the WSH engine. I called this file runvsnet.js.

```
var WshShell = WScript.CreateObject("WScript.Shell");

function Control() {
    var dte, docs, doc;
    dte = GetObject("", "VisualStudio.DTE");
    docs = dte.Documents;
    docname = "";
    for (i = 1; i <= docs.Count; i++) {
        doc = docs.Item(i);
        docname = docname + doc.FullName + "\n";
    }

    WshShell.Popup(docname);
}

Control();
```

You can run this file just as you did the runvsnet.vbs file, by double-clicking it, by running it directly, or by launching it in the wscript program.



The Windows Script Host engine has two entry points, the `wscript.exe` program and the `cscript.exe` program. The only difference is that `cscript` opens a console window so that your script can write to the console, whereas `wscript` does not. You can use either to run your scripts, based on whether you need console output or not.

Delphi

My heart is in Delphi. I began using the language the day version 1.0 was released, and to this day I think it is a beautiful language, especially after I wrote a few books about it. But readers of computer books don't want to hear this kind of thing, so let me get to the point here: If you like Delphi, you can use it to write your automation clients.

Interestingly, I chose to use an earlier version of Delphi (Delphi 3.0) for this sample, so that regardless of which version you have (7.0 is current as of this writing), you should be able to use this example. To create this project, follow these steps carefully. The form you're developing is shown in Figure 15.1.

1. Create a new application, and resize the form so its `Width` is about 500 pixels and its `Height` is about 250 pixels. (You don't have to be exact.)
2. Drop a Panel on the form. Set its `Align` property to `alTop`. Set its `Caption` property to an empty string, and set its `BevelOuter` property to `bvLowered`.
3. Drop a Button on the Panel (not on the main part of the form; the Button must be a child of the Panel). Put the Button toward the left edge, as shown in Figure 15.1. Set the Button's `Caption` property to `Docs`.
4. Drop a Memo on the form below the Panel. (*Don't* put the Memo on the Panel.) Set the Memo's `Align` property to `alClient`. This will cause the Memo to fill the part of the form not taken up by the panel, allowing you to resize the Memo by resizing the form.
5. Now double-click the button and add the code shown in the `Button1Click` handler in the code below.

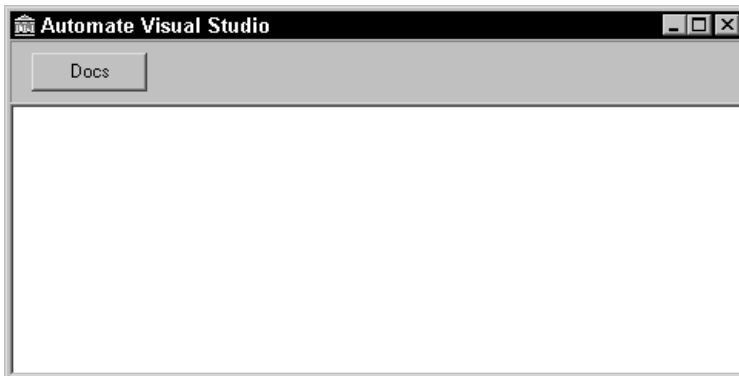


Figure 15.1 The Delphi form has a Panel, a Button, and a Memo control.

Following is the code for the entire unit:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Button1: TButton;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses ComObj;

procedure TForm1.Button1Click(Sender: TObject);
var
  dte, docs, doc: Variant;
  i: Integer;
begin
  try
    dte := GetActiveOleObject('VisualStudio.DTE');
  except
    dte := CreateOleObject('VisualStudio.DTE');
  end;
  docs := dte.Documents;
  for i := 1 to docs.Count do
  begin
    doc := docs.Item(i);
    Memo1.Lines.Add(doc.FullName);
  end;
end;

end.
```

Notice how I obtained the object in this code: First, I had to use the `ComObj` unit. Next, I called `GetActiveOleObject` in an attempt to get the currently running instance of Visual Studio .NET. (If one isn't running, the `GetActiveOleObject` function will throw an exception, rather than return a null object.) Thus, I provide an exception handler that instead calls `CreateOleObject`, which is the Delphi function for creating a new COM object.

The Delphi COM objects are instances of type `Variant`, which tells the Delphi compiler to allow any members for the variables, recognizing that the COM system will determine at runtime whether the members are valid. Thus, unlike Visual Studio .NET, you won't see the members pop up. (Delphi does provide extended ways to import a COM object and generate a Delphi unit for the COM object, but this way is easier.) You can see that I simply accessed the members of the `DTE` object, starting with `Documents`. From there I accessed the items just as I did in the WSH examples in the section, "Windows Script Host." But instead of populating a string, I populated the memo on the form.

When you run this program, you will get a form that looks just like the one you laid out at design time. When you click the button, you will see a list of the open documents, just as you did in the sample WSH scripts.

Python

Python is an amazing language. Although it's considered a script language, it has several wonderful features, such as dynamic classes. This means you can add new members to an object at runtime, which makes for an extremely powerful language. Not many languages have this feature.

If you download the version of Python created by ActiveState Corporation (www.activestate.com), you also get a module that allows your Python programs to interact with the COM system. The Python language includes a command-line interface that you can use for testing your COM objects on the fly. Here's a sample session, including an error I encountered because I couldn't remember whether Python has predefined names for true and false (it doesn't):

```
C:\Program Files\ActivePython22>python
ActivePython 2.2.1 Build 222 (ActiveState Corp.) based on
Python 2.2.1 (#34, Apr 15 2002, 09:51:39) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import win32com.client
>>> obj = win32com.client.Dispatch("VisualStudio.DTE")
>>> doc = obj.ActiveDocument
>>> doc.FullName
u'C:\\\\dev\\Chapter13\\CSharpConsoleTest\\Class1.cs'
>>> doc.Kind
u'{8E7B96A8-E33D-11D0-A6D5-00C04FB67F6A}'
>>> sel = doc.Selection
>>> sel.LineDown(true)
```

```

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'true' is not defined
>>> sel.LineDown(1)
>>> sel.Text
u'    public static void Main()\r\n'
>>>

```

You can see in the first line that I started up the python command-line interface. Then I *imported* the `win32com.client` library, giving me COM capabilities. Next, I located the existing instance of the Visual Studio .NET's DTE object using the `Dispatch` function, and I saved the object in the `obj` variable. Notice that I didn't provide a type name for the `obj` variable; Python is not a strongly typed language like C++. (This loosely typed feature allows you to create a single array or list and place any objects in the array or list, regardless of type.)

After obtaining the DTE object, I grabbed the `ActiveDocument` object, saving it in the `doc` variable (again, without having to specify any type information). I then retrieved the `FullName` member of the `doc` variable. When you call a function that returns a value, or simply obtain a value as I did by retrieving the `FullName` member, the Python command-line interpreter immediately prints the results to the screen. You can see that it printed out:

```
u'C:\\dev\\Chapter13\\CSharpConsoleTest\\Class1.cs'
```

The `u` at the beginning indicates this is a Unicode string. Python encloses strings in single quotes, but the language lets you choose if you want to surround a string in single or double quotes. (That way, if you have to put quotes inside the string, you can often get away with not escaping them; you just use the other type of quotes to surround the string, assuming you don't have the other type of quotes also inside the string. Otherwise, you have to escape them as you do in C++ and other languages.)

Next I obtained the `doc.Kind` variable just to see what I would get back. As I expected, I got back the string for the GUID. Since, by default, Python on WinNT-based systems (such as 2000 and XP) uses Unicode, I once again got back a Unicode string.

Remember, this is all happening in real time. As I type a line, the interpreter interacts with Visual Studio .NET and immediately performs the task. This became apparent in the next steps I took, as I interacted with the editor. In the next line I obtained the `Selection` object and saved it to the `sel` variable. The `Selection` object includes a member function called `LineDown` that takes a single parameter, a Boolean variable, specifying whether to select the text while moving down or to just move the cursor down. (An overloaded version of `LineDown` takes a second parameter, the number of lines to move down. The version I used here moves down just one line.) I wanted to select the text, so I was hoping to pass a `true`. However, I forgot that Python does not have Boolean variables (which has caused some lively debates on the online boards). So when I tried passing `true`, I received an error. (The error message indicated that the first line of my standard input had an error.) So I tried the statement again (by pressing

the up arrow key), but this time passed a 1, and it worked. As soon as I pressed Enter, I saw in the Visual Studio .NET window (which I had carefully positioned so I could see it alongside my Python command-line window) that the cursor had moved down, and that the line it was on previously became selected—all in real time!

Finally, I obtained the `Text` property of the `sel` object, and Python printed the results to the screen. What appeared was exactly the text that was selected in the Visual Studio .NET editor.

Next check out this simple script that I called `runvsnet.py`. I created this using a text editor and saved it with the `.py` extension. This script does the same thing as the previous WSH scripts and Delphi program, but instead of displaying the document names in a window, it prints them out to the console window. After you look at the script, I have a few comments to share before I show you how to run it. (Type this script into an editor, and save the file as `runvsnet.py`.)

```
import win32com.client

class Controller:
    def __init__(self):
        self.DTE = win32com.client.Dispatch("VisualStudio.DTE")

    def ListDocuments(self):
        docs = self.DTE.Documents
        for doc in docs:
            print doc.FullName

print "===Documents==="
c = Controller()
c.ListDocuments()
```

As you can see, rather than using braces, Python relies on the indentations to determine when blocks finish. Thus, the class declaration ends with the line `print doc.FullName`. Python also runs the outermost lines first in order. Thus, for this code, it first runs the `import` statement; next it processes the class block (that is, it simply learns about the `Controller` class), then the `print` statement, then the `c =` assignment, and finally the `c.ListDocuments()` line. Notice also that to create a new object, Python has no `new` keyword. Instead, you just call the class like a function.

The following line runs the python interpreter. (I was in the directory containing the python interpreter. Alternatively, you can add the interpreter's directory to your path.)

```
C:\Program Files\ActivePython22>python runvsnet.py
```

Here's the output after I ran the preceding line:

```
===Documents===
C:\Write2001\VS.NET-IDE\dev\Chapter13\OfficeVSAddin\Connect.vb
File System (CSharpGeneralSetup)
```

```
C:\Write2001\VS.NET-IDE\dev\Chapter13\CSharpGeneral\Connect.cs
C:\Write2001\VS.NET-IDE\dev\Chapter13\TestData.txt
C:\Write2001\VS.NET-IDE\dev\Chapter13\CSharpConsoleTest\Class1.cs
```

Here's another, slightly modified, Python script. This one matches the functionality of the sample VBScript and JScript scripts shown in the "Windows Script Host" section earlier in this chapter.

```
import win32com.client
import win32gui

class Controller:
    def __init__(self):
        self.DTE = win32com.client.Dispatch("VisualStudio.DTE")

    def ListDocuments(self):
        text = ""
        docs = self.DTE.Documents
        for doc in docs:
            text = text + doc.FullName + "\n"
        win32gui.MessageBox(0, text, "Documents", 0)

c = Controller()
c.ListDocuments()
```

I called this script `runvsnet2.py`. When you run this script like so:

```
C:\Program Files\ActivePython22>python runvsnet2.py
```

you will see a message box appear, listing the names of the currently opened documents in Visual Studio .NET. You can see how I called the `MessageBox` function. This is part of the Win32 API (which you may or may not be familiar with depending on your age). The Win32 API functions that deal with the windowing system are in the `win32gui` Python module. Thus, I import the `win32gui` module and then fully qualify the `MessageBox` function name as `win32gui.MessageBox`. The first parameter is the handle to an existing window that will serve as a parent to the message box (I always just pass 0, meaning no parent). The second parameter is the message to display; the third parameter is the title bar for the message box. The final parameter is a constant representing the buttons to include on the message box; 0 means a single OK button.

Script Explorer Add-in

So far in this chapter I've shown you that you can automate Visual Studio .NET using pretty much any language you want, provided that language has access to the COM system. Therefore, it seemed logical that Visual Studio .NET should have a tool window

similar to the Macro Explorer that lists scripts in other languages, enabling you to double-click the scripts in this window to run them.

The add-in I describe here does just that. It has a tool window containing a treeview control that lists the scripts in a given directory. If you double-click the script, the script runs. Like many of the previous add-ins, this one uses the VSUserControlHost that I described in “Using the Form Designer with a Tool Window” in Chapter 7. And as for the other add-ins, be sure to add a COM reference to the VSUserControlHost 1.0 Type Library.

Here’s the Connect module for the add-in:

```
Imports Microsoft.Office.Core
imports Extensibility
imports System.Runtime.InteropServices
Imports EnvDTE

<GuidAttribute("F3D6C34F-CCDB-4D03-94D3-C9E5A15A9491"), _
ProgIdAttribute("ScriptExplorer.Connect")> _
Public Class Connect

    Implements Extensibility.IDTExtensibility2
    Implements IDTCommandTarget

    ' Note: I changed applicationObject to public shared!
    Public Shared applicationObject As EnvDTE.DTE
    Dim addInInstance As EnvDTE.AddIn
    Private doc As VSUserControlHostLib.IVSUserControlHostCtl = Nothing
    Private toolwin As Window = Nothing

    Public Sub OnBeginShutdown(ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnBeginShutdown
    End Sub

    Public Sub OnAddInsUpdate(ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnAddInsUpdate
    End Sub

    Public Sub OnStartupComplete(ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnStartupComplete
    End Sub

    Public Sub OnDisconnection(ByVal RemoveMode As _
    Extensibility.ext_DisconnectMode, ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnDisconnection
    End Sub

    Public Sub OnConnection(ByVal application As Object, _
    ByVal connectMode As Extensibility.ext_ConnectMode, _
    ByVal addInInst As Object, ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnConnection
```

```

applicationObject = CType(application, EnvDTE.DTE)
addInInstance = CType(addInInst, EnvDTE.AddIn)
Dim tempdoc As Object

Dim newguid As String = "{779C198F-F4AC-4d21-A06A-EF710D3946A5}"
toolwin = applicationObject.Windows.CreateToolWindow( _
    addInInstance, _
    "VSUserControlHost.VSUserControlHostCtl", _
    "Scripts", newguid, tempdoc)
toolwin.Visible = True
doc = CType(tempdoc, VSUserControlHostLib.IVSUserControlHostCtl)
Dim asm As System.Reflection.Assembly
asm = System.Reflection.Assembly.GetExecutingAssembly()
doc.HostUserControl(asm.Location, _
    "ScriptExplorer.ScriptExplorerForm")
Try
    Dim commands As Commands = applicationObject.Commands
    Dim command1 As Command = commands.AddNamedCommand( _
        addInInstance, _
        "Show", "Script Explorer", "Shows the Script Explorer", _
        True, 59, Nothing, _
        vsCommandStatus.vsCommandStatusSupported + _
        vsCommandStatus.vsCommandStatusEnabled)
    Dim viewMenu As CommandBarPopup = _
        applicationObject.CommandBars("MenuBar"). _
        Controls("&View")
    Dim viewMenuBar As CommandBar = viewMenu.CommandBar
    Dim othersMenu As CommandBarPopup = _
        viewMenu.Controls("Other Windows")
    Dim othersBar As CommandBar = othersMenu.CommandBar
    command1.AddControl(othersBar, 1)

Catch
End Try
End Sub

Public Sub Exec(ByVal cmdName As String, ByVal executeOption As _
    vsCommandExecOption, ByRef varIn As Object, ByRef varOut As Object, _
    ByRef handled As Boolean) Implements IDTCommandTarget.Exec
    handled = False
    If (executeOption = vsCommandExecOption. _
        vsCommandExecOptionDoDefault) Then
        If cmdName = "ScriptExplorer.Connect.Show" Then
            toolwin.Visible = True
            handled = True
            Exit Sub
        End If
    End If
End Sub
End Sub

```

```

Public Sub QueryStatus(ByVal cmdName As String, ByVal neededText _
As vsCommandStatusTextWanted, ByRef statusOption As _
vsCommandStatus, ByRef commandText As Object) _
Implements IDTCommandTarget.QueryStatus
    If neededText = EnvDTE.vsCommandStatusTextWanted. _
vsCommandStatusTextWantedNone Then
        If cmdName = "ScriptExplorer.Connect.Show" Then
            statusOption = CType(vsCommandStatus. _
vsCommandStatusEnabled + vsCommandStatus. _
vsCommandStatusSupported, vsCommandStatus)
        Else
            statusOption = vsCommandStatus.vsCommandStatusUnsupported
        End If
    End If
End Sub
End Class

```

Now here's the form, as shown in Figure 15.2. To create this form, follow these steps:

1. Add a new User Control.
2. Place a StatusBar control on the form.
3. Place a TreeView control on the form.
4. Set the TreeView's Dock property to Fill. Make sure its ShowPlusMinus property is True, and its ShowRootLines property is True.
5. Click the Nodes property to select it, then click the button with an ellipses (...) on it to open the TreeNode Editor, which is shown in Figure 15.3.
6. In the TreeNode Editor, click the Add Root button. Then type the word Scripts into the Label box. Click OK to close the TreeNode Editor.
7. In the Toolbox, double-click the ContextMenu control to add a context menu to the form.
8. Click the ContextMenu control in the component tray beneath the form; the menu editor will open, as shown in Figure 15.4. Add the names Choose directory and Edit, as shown in the figure.

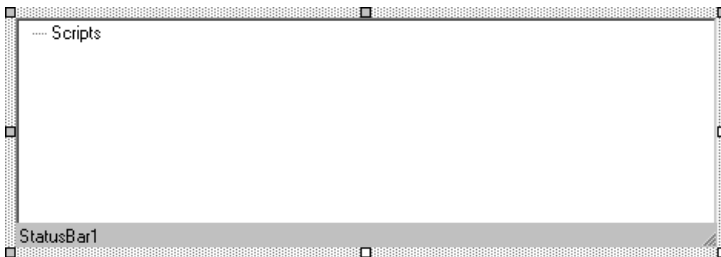


Figure 15.2 The form for the add-in has a StatusBar and a TreeView control.

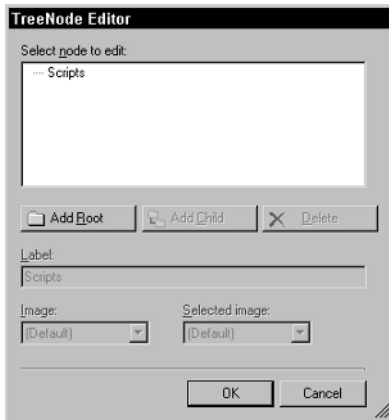


Figure 15.3 Use the TreeNode Editor to add a root node to the TreeView control.

9. Now switch to the code editor by right-clicking on the form, and in the popup menu choosing View Code.
10. In the drop-down list in the upper left of the code editor, choose MenuItem1. Then in the drop-down list on the right, choose Click to add a new Click handler for the menu item, as shown in Figure 15.5. Then do the same to add handlers for the DoubleClick event of TreeView1 and the Click event for MenuItem2.

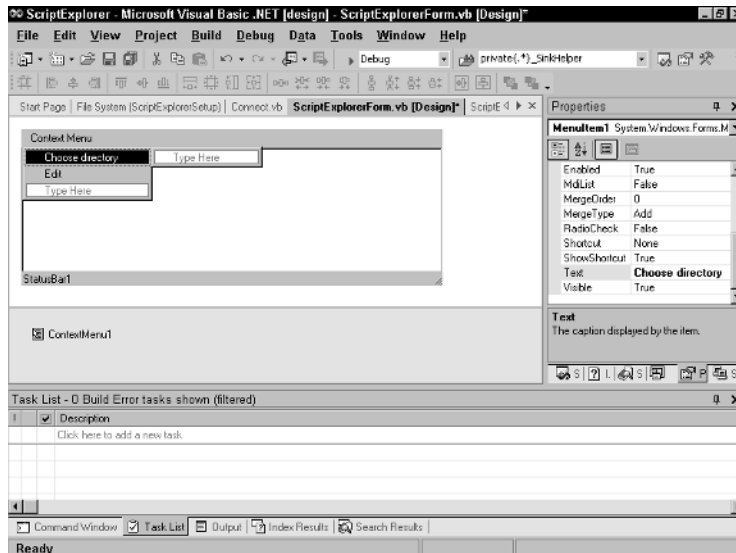


Figure 15.4 Use the Menu Editor to add two menu items.

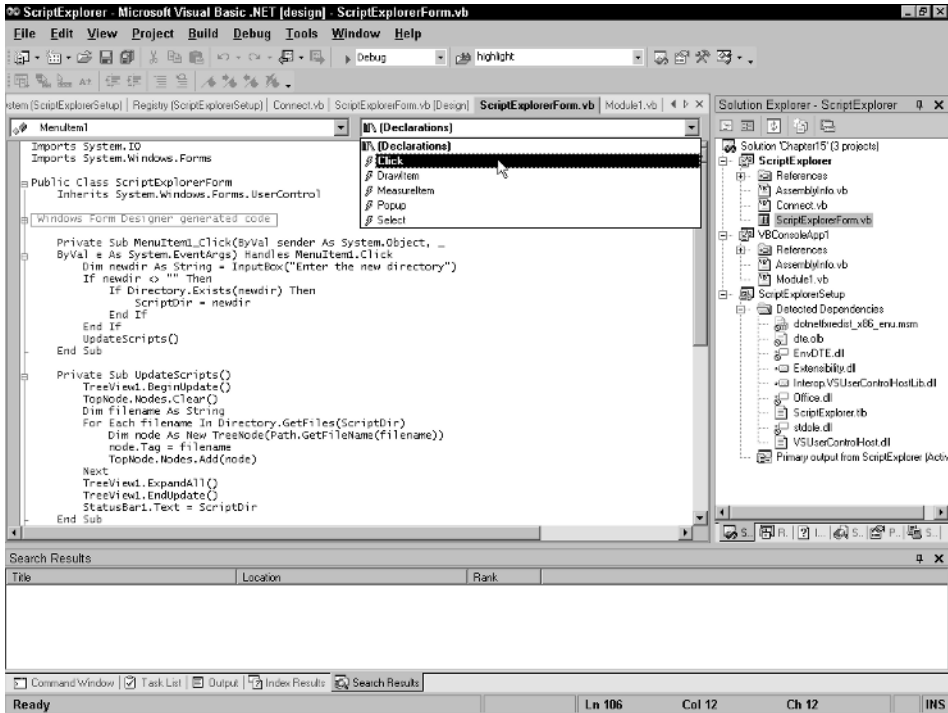


Figure 15.5 The drop-down listboxes let you add event handlers.

- Now enter the following code, some of which will already be present, so you won't have to type it in. Specifically, notice that I added some initialization code to the New constructor; I also added the event handlers after the #End Region statement.

```
Imports System.IO
Imports System.Windows.Forms

Public Class ScriptExplorerForm
    Inherits System.Windows.Forms.UserControl

    #Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call
        TopNode = TreeView1.Nodes.Item(0)
    End Sub

    #End Region
End Class
```

```

        If Not Directory.Exists(ScriptDir) Then
            Directory.CreateDirectory(ScriptDir)
        End If
        UpdateScripts()
    End Sub

    'UserControl overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As
Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.IContainer

    Friend WithEvents TreeView1 As System.Windows.Forms.TreeView
    Friend WithEvents ContextMenu1 As
System.Windows.Forms.ContextMenu
    Friend WithEvents MenuItem1 As System.Windows.Forms.MenuItem
    Friend WithEvents StatusBar1 As System.Windows.Forms.StatusBar
    Friend WithEvents MenuItem2 As System.Windows.Forms.MenuItem
    <System.Diagnostics.DebuggerStepThrough()> Private Sub _
InitializeComponent()
        Me.TreeView1 = New System.Windows.Forms.TreeView()
        Me.ContextMenu1 = New System.Windows.Forms.ContextMenu()
        Me.MenuItem1 = New System.Windows.Forms.MenuItem()
        Me.StatusBar1 = New System.Windows.Forms.StatusBar()
        Me.MenuItem2 = New System.Windows.Forms.MenuItem()
        Me.SuspendLayout()
    End Sub

    'TreeView1
    '
    Me.TreeView1.ContextMenu = Me.ContextMenu1
    Me.TreeView1.Dock = System.Windows.Forms.DockStyle.Fill
    Me.TreeView1.ImageIndex = -1
    Me.TreeView1.Name = "TreeView1"
    Me.TreeView1.Nodes.AddRange(New
System.Windows.Forms.TreeNode() _
        {New System.Windows.Forms.TreeNode("Scripts")})
    Me.TreeView1.SelectedIndex = -1
    Me.TreeView1.Size = New System.Drawing.Size(456, 150)

```



```
Me.TreeView1.TabIndex = 0
,
'ContextMenu1
,
Me.ContextMenu1.MenuItems.AddRange(New System.Windows.Forms. _
    MenuItem() {Me.MenuItem1, Me.MenuItem2})
,
'MenuItem1
,
Me.MenuItem1.Index = 0
Me.MenuItem1.Text = "Choose directory"
,
'StatusBar1
,
Me.StatusBar1.Location = New System.Drawing.Point(0, 134)
Me.StatusBar1.Name = "StatusBar1"
Me.StatusBar1.Size = New System.Drawing.Size(456, 16)
Me.StatusBar1.TabIndex = 1
Me.StatusBar1.Text = "StatusBar1"
,
'MenuItem2
,
Me.MenuItem2.Index = 1
Me.MenuItem2.Text = "Edit"
,
'ScriptExplorerForm
,
Me.Controls.AddRange(New System.Windows.Forms.Control() { _
    Me.StatusBar1, Me.TreeView1})
Me.Name = "ScriptExplorerForm"
Me.Size = New System.Drawing.Size(456, 150)
Me.ResumeLayout(False)

End Sub

#End Region

Private Sub MenuItem1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MenuItem1.Click
    Dim newdir As String = InputBox("Enter the new directory")
    If newdir <> "" Then
        If Directory.Exists(newdir) Then
            ScriptDir = newdir
        End If
    End If
End Sub
```

```
        End If
        UpdateScripts()
    End Sub

    Private Sub UpdateScripts()
        TreeView1.BeginUpdate()
        TopNode.Nodes.Clear()
        Dim filename As String
        For Each filename In Directory.GetFiles(ScriptDir)
            Dim node As New TreeNode(Path.GetFileName(filename))
            node.Tag = filename
            TopNode.Nodes.Add(node)
        Next
        TreeView1.ExpandAll()
        TreeView1.EndUpdate()
        StatusBar1.Text = ScriptDir
    End Sub

    Private Sub TreeView1_DoubleClick(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles TreeView1.DoubleClick
        Dim node As TreeNode = TreeView1.SelectedNode
        If Not node Is Nothing Then
            If node Is TopNode Then Exit Sub
            Dim procinfo As New ProcessStartInfo()
            procinfo.UseShellExecute = True
            procinfo.FileName = node.Tag
            Dim proc As Process = Process.Start(procinfo)
        End If
    End Sub

    Private Sub MenuItem2_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MenuItem2.Click
        Dim node As TreeNode = TreeView1.SelectedNode
        If Not node Is Nothing Then
            If node Is TopNode Then Exit Sub
            Connect.applicationObject.ItemOperations.OpenFile(node.Tag)
        End If
    End Sub

    Private ScriptDir As String = "c:\scripts"
    Private TopNode As TreeNode = Nothing

End Class
```

When you build this add-in and run it, Visual Studio .NET will now have a tool window that lists scripts. I put all my scripts in a single directory called `c:\scripts`, which this tool window lists by default. If you double-click a script in the list, the script will run. If you right-click the script's name, you can choose Edit to edit the script right inside the Visual Studio .NET IDE. Or you can right-click and type in a new directory. (You're welcome to enhance the form to include an `OpenFile` dialog to choose a different directory.)

As you can see, this add-in simply executes whichever file you double-click. Thus, you can also use this add-in to spawn other programs besides scripts. And if you're adventurous, you can play with the `procinfo` and `Process.Start` items in the `TreeView1.DoubleClick` event handler to capture the standard output and standard error, writing it to a tool window. (If you do this, drop me a line at readers@jeffcogswell.com; I'll add a couple of the ones that work to the official Web site for this book. And if you have other additions, don't hesitate to share them with me and the other readers in the forum on the book's Web site.)

Moving Forward

This chapter showed you how you can take the automation object exposed by Visual Studio .NET and control the IDE from pretty much any program that supports COM automation. To demonstrate this capability, I used Delphi, Python, and the Windows Script Host, or WSH. I then took you through an interesting add-in that lets you spawn these other programs so you can, effectively, write your macros in any language you want.

The next chapter, "Deploying Your Macros and Add-ins," begins Part IV, "Deploying and Supercharging." In it, I talk about the various issues regarding the deployment of your add-ins to other computers, focusing in particular on some important points regarding Office add-ins. In the final chapter of this book, Chapter 17, "Supercharging Visual Studio .NET," I close some open ends about add-ins and macros.

PART

Four

Deploying and Supercharging

TEAMFLY

Deploying Your Macros and Add-ins

Deploying add-ins is easy, because when you use the Add-in wizard, you automatically get a Setup project, which you can provide to your users. That said, some security issues come into play if your add-ins are for Microsoft Office. In the first of two main sections in this chapter I discuss these security issues. In the second, I show you how you can easily deploy your macros to your end users.

All about Security and Add-ins

Security is an important aspect of all software, not just add-ins. By security, I mean whether or not a program that's running is allowed, for example, to access important system files such as those in the C:\Windows\System32 directory. Security can also mean whether a program can modify the system Registry, and if so, which portions of it.

When dealing with .NET, security issues are handled on two levels:

Operating system level. The operating system limits various users from performing certain operations based on the access granted by the system administrator. Windows NT, 2000, and XP all provide such security measures. Systems administrators can prevent, for example, various users from running programs that write to the Windows directory.

.NET system level. The .NET system includes additional security described by Microsoft as “finer-grained” than at the operating system level. The .NET system can determine, for example, that one .NET program can create files on the local disk drive, while another program cannot, even though the user who is logged on to the computer may have full permissions to write to the hard drive.

In addition to these two levels, security has two sides:

System administration side. This refers to the security levels the user or systems administrator sets for the computer.

Application side. The security access an application requires to run.

In the following sections I provide you with information on software and security, in particular how it pertains to add-ins and macros. An understanding of this information is essential to have before you attempt to deploy an add-in or macro on another user's computer. The last thing you want to do is make an add-in available to hundreds of users, only to find out that none of the users can run the add-in because it lacks the needed security privileges.

.NET Security

Before delving into the details of .NET security, you must thoroughly understand what is meant by *managed code*. When you run managed code under .NET, you load code in the form of Microsoft Intermediate Language (MSIL). MSIL code is a low-level byte-code that can easily be translated to native machine code. The MSIL code lives in an executable file that also contains high-level information, called *metadata*, which describes the code and its data structures. The .NET runtime is then able to carefully inspect the code and make sure it is safe. For example, if you download code from the Internet, and you're not sure the code behaves the way it's expected to (such as whether the code will attempt to delete important system files or upload your address book to a Web site), the .NET runtime can inspect the code and make sure it doesn't perform any secret, dangerous tasks. In other words, the .NET runtime manages the code before and while it runs. All calls the program makes into the operating system are watched by the .NET system. If the program attempts to open a file, for example, the program calls the static (or shared) members of the `File` class, which is part of the .NET framework. The program does not call directly into the operating system. (Of course, you can write .NET code in C# and VB.NET that calls directly into the operating system, but to do that, you have to use unmanaged code, which is untrusted.)

In addition, managed code implements managed objects. These managed objects are allocated in the managed heap, rather than a program's own heap. While in the managed heap, the .NET system can watch over them and control them, performing such tasks as deleting them when they are no longer used. (That's why you don't have to delete your managed objects; the Common Language Runtime, or CLR, does it for you.)

Thus, a program can have two types of code:

Managed. This is code that runs under the .NET system, including the Common Language Runtime.

Unmanaged. This is straight machine code that does not run under the .NET system and its Common Language Runtime.

Since one main feature of .NET is distributed programming, users can download managed .NET code from the Internet and run it locally on their own computer. Certainly,

the users don't want any potentially damaging or dangerous code to run; thus, the .NET system includes strict security under which managed code runs. (Unmanaged code, when not used in conjunction with other security tools such as Microsoft Authenticode, is by default considered untrusted and is akin to downloading some strange .EXE file attached to an email message sent by someone you don't know with a subject like "My party was a blast! Run this program to see!")

Valid and Verified .NET Code

When you run a .NET program, the .NET system loads the program from an assembly (which is just an .EXE or a .DLL file). Since the code exists in the form of intermediate language (MSIL), the code must be compiled by the Just-in-Time (JIT) compiler. But before compiling the code, the .NET system first inspects the code to make sure it is:

Valid. The .NET system makes sure the code is legitimate. For example, the system makes sure the code doesn't contain incorrect bytes that cannot be compiled, and that the code conforms to valid MSIL grammar; in other words, validation operates at the *syntax level*.

Verified. This means the code does what it claims to do. In other words, verification operates at the *semantic level*.

To make sure the code is valid, the JIT compiler inspects it and looks for errors. Verifying the code is considerably more complex, and to do the verification the .NET system uses a complex algorithm. This involves processes such as making sure objects are initialized before being used (remember, such code may be syntactically correct but not semantically correct) and making sure the code doesn't use pointers to reach into the protected areas of objects.

As a first step, then, before managed code can be considered trusted, it must be confirmed as both *valid* and *verified*. In addition, the .NET system also makes sure the metadata is valid, that is, that it makes sense and is legitimate. Only then does the system run the JIT compiler to generate machine code. But even after the machine code is generated, remember that such code still calls into the .NET framework to perform its operating system requests such as file creation and opening. Thus, the code, although now in machine format, is still *managed*.

Security Permissions

In addition to checking that the code is valid and verified, the .NET system also determines whether a program is allowed to have access to various system resources such as files and the system Registry. To do so, the .NET system gathers up various pieces of information about the code that is trying to run, including who created it, where it came from, and, if it came from the Internet, which site, and whether it has a Microsoft Authenticode certificate. The .NET system collects all this information and produces a security level for the .NET program.



When you deploy a .NET program onto another computer, although you can let the computer determine the security level of your program, your program can also request a certain security level. For example, if you know that your program will need to write to the administrator-only sections of the Registry, you can write code in your program stating that your program will need this high level of security. To find out more about how to do this, open the .NET online help, and in the index type “declarative security syntax.” Click the entry, and in the Index results, choose Declarative Security.

After obtaining a security level, the .NET system checks which processes the local computer allows programs with that security level to perform. These are configurable by the system administrator (or, in the case of home computers, the user him- or herself). The .NET system then determines whether the .NET program is allowed to perform the requested tasks.

These tasks include the capability to read, write, create, and append files, to print to the printers, to call into unmanaged code, to skip code verification, to access system services, to access user-interface functionality, to make and accept socket (i.e., Internet) connections, to modify the Registry, and to access the system logs.



To see the complete list of tasks that require permission, open the .NET online help, and type “code access security” into the index. Under the Code Access Security heading, click the item called Permissions.

Security Administration for .NET

The user or system administrator of a computer decides which permissions to grant to programs; however, the .NET installation program creates a set of defaults that are sufficient for most situations. To administer such security grants for the .NET system, you use the .NET Framework Configuration program, shown in Figure 16.1.

You can access this program from three places:

- Directly open C:\Windows\Microsoft.NET\Framework\v1.0.3705\mscorcfg.msc. (Version 1.0.3705 was current at the time of this writing.) The file has an .msc application, which means it will open inside the program C:\Windows\System32\mmc.exe, the Microsoft Management Console.
- From the Windows desktop choose Start⇨Programs⇨Administrative Tools⇨Microsoft .NET Framework Configuration.
- Open the Control Panel, double-click Administrative Tools, then double-click Microsoft .NET Framework Configuration.

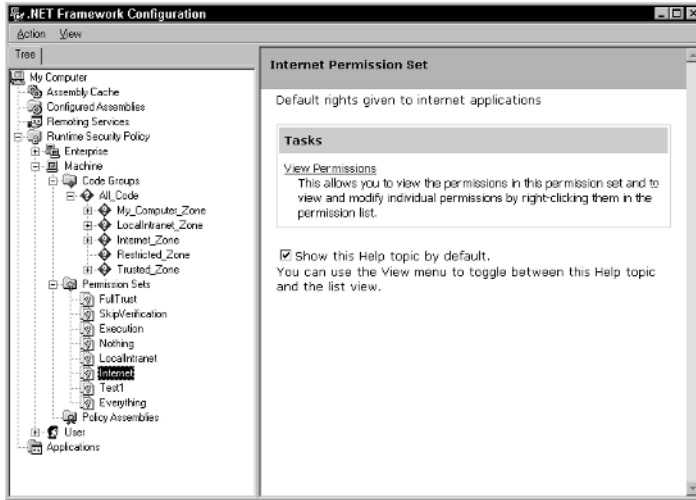


Figure 16.1 .NET Framework Configuration Program.

Security in Action

If you want to see .NET security in action, open your favorite text editor, type in the following program, and save it with the filename `security.cs`.

```
using System;
using System.IO;
public class printClass
{
    public static void Main()
    {
        StreamWriter f = File.CreateText("c:\\security.txt");
        f.WriteLine("Hello");
        f.WriteLine(System.DateTime.Now);
        f.Close();
    }
}
```

This program is rather benign; it simply opens a file called `security.txt` in the root directory of the C: drive and writes the string "Hello" followed by the current date and time. Then it closes the file.

Compile the file by typing this at a DOS prompt:

```
csc security.cs
```

If there are no typos, you should see no compiler errors. Then you can run the file:

```
security
```

Look at the root directory of your C: drive, where you should see the security.txt file, meaning the program worked. But now try this: If you have access to Web space, upload the executable file, security.exe, to it. Then try to open the file in your browser. I uploaded mine to the root of my www.jeffcogswell.com Web server; then went to my Web browser and typed this:

```
http://www.jeffcogswell.com/security.exe
```

You would, of course, substitute whichever Web server you chose. Internet Explorer will download the file and look it over. When it detects that it's a .NET file, it will begin running the program—without first asking whether it should (as it normally would with an executable file). However, the program is managed, so the .NET system will be watching it carefully to make sure the program doesn't do anything it's not supposed to do, which includes writing to the local disk drive. And when the CreateText function occurs, the .NET system will throw an exception, either System.Security.SecurityException or System.Security.Policy.PolicyException, depending on your operating system, as shown in Figure 16.2.



If you decide to try this experiment—that is, putting the security.exe file on your own Web server—do not put it on a secure (https) site. Microsoft has acknowledged a bug wherein Internet Explorer may shut down or you will get a FileNotFound exception. Refer to Microsoft Knowledge Base article number 312546 for more information.

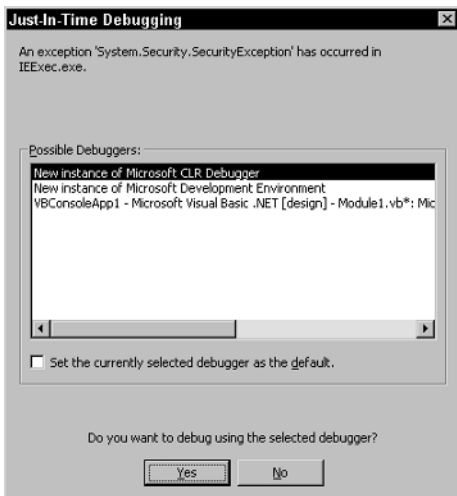


Figure 16.2 The program runs, but throws an exception.

If you then start the debugger (I usually use the Microsoft CLR Debugger, simply because it loads quickly), you will see the full exception information, as shown in Figure 16.3. In the output window in the debugger, you can see the full information for the error. Depending on the operating system version, your message will be something like the message shown in Figure 16.3

The error was due to lack of permission of type FileIOPermission. This is no surprise, since the program was written specifically to write to the disk drive. The program that generated the error was IEEExec.exe. This program is part of the .NET system, and is called the Microsoft IE Execute Shell. The full path to the program is C:\WINNT\Microsoft.NET\Framework\v1.0.3705\IEExec.exe.

By default, here are the permissions that are allowed by a program that you run from the Internet:

File Dialog. Open

Isolated Storage File. Domain isolation by user; disk quota 10240

Security. Enable Code Execution

User Interface. Safe top-level windows; own clipboard

Printing. Safe printing

To see which permissions are set on your computer, open the .NET Framework Configuration utility and drill down to My Computer→Runtime Security Policy→Machine→Permission Sets→Internet. In the right-hand pane you will see the permission groups. Double-click to see the individual permissions.

Here's the source code for another test you can try:

```
using System;
using System.IO;
public class printClass
{
    public static void Main()
    {
        System.Windows.Forms.MessageBox.Show(
            "Hello from C#", "Security2");
    }
}
```



Figure 16.3 You can see the full exception information inside the debugger.

This code opens a top-level window, which is allowed. If you compile this and upload it (I put it at www.jeffcogswell.com/security2.exe), then, depending on the basic security levels you choose for Internet Explorer, Internet Explorer will simply run the program, *no questions asked*. But if this makes you uncomfortable, you can change it. You can either set up the .NET system so it won't run such code at all, or you can limit the permissions even further, such as by revoking permission to display a user interface. (You could, I suppose, grant additional permissions, but I don't recommend that, as that would allow untrustworthy code to do the things you grant permission for, such as write to the hard drive.

COM Security

When you develop an add-in, remember, you are developing a .NET assembly *and* registering it as a COM component. The COM component is, in turn, an add-in to either Visual Studio .NET or an Office application. In the case of Office applications, the add-in is subject to the security imposed by the Office products. *This security system is completely independent of the .NET security system.*

To see the security settings, start Microsoft Word (or any Office application) and choose Tools→Macro→Security. You will be presented with three choices: high, medium, and low. Most system administrators prefer to set the security to high, which forbids any nondigitally signed add-ins and macros from running. When running at the medium level, the Office application prompts the user before running the add-in. In low security, the application runs all add-ins.

This presents a problem for the add-ins that are also .NET-managed assemblies. Because the add-in is actually an assembly, not directly a COM component, when you install a managed add-in, the shared add-in wizard registers the file `mscorlib.dll` (normally found in `C:\Windows\System32`) as the COM file. The add-in is registered with the COM system as a COM component, which has its own GUID listed in with the classes in the Registry; however, the server is given as `mscorlib.dll`. For example, here are the keys in the Registry from a test add-in that I wrote:

```
HKEY_CLASSES_ROOT
  CLSID
    {24224495-7902-494B-B749-E3247D233BF1}
      Implemented Categories
      InprocServer32
      ProgId
```

The `InprocServer32` entry, in turn, has these named values:

```
(default)="C:\\WINNT\\System32\\mscorlib.dll"
ThreadingModel="Both"
Class="OfficeVSAddin2.Connect"
Assembly="OfficeVSAddin2, Version=1.0.1063.30401,
Culture=neutral, PublicKeyToken=null"
RuntimeVersion="v1.0.3705"
CodeBase="file:///C:/dev/OfficeVSAddin2/bin/OfficeVSAddin2.DLL"
```

The default item is the actual server that the Office product calls into. The `mscoree.dll` file, in turn, looks in the Registry to determine the assembly information. The `mscoree.dll` file uses the .NET interop features to expose the assembly's `IDTEx-tensibility2` interface to the Office product, allowing the add-in to run.

And that's where the problem lies: The add-in itself is separate from the COM component. The Office application doesn't care which files the COM component ultimately uses to obtain its code; all the Office application cares is that the COM component *itself* is digitally signed. And guess what? The `mscoree.dll` file is *not* digitally signed; it can't be, because the digital signature is affiliated with a person or organization, and all people and organizations developing managed add-ins share this same `mscoree.dll` file (unless they create their own unmanaged COM component). And since this file is not signed, when Office runs in high security mode, it will refuse to let the `mscoree.dll` add-in run. (Remember, Office considers the COM component itself the add-in, even though you and I both know the real add-in is in *your* assembly that `mscoree.dll` calls into.) What does this mean? When Office runs in high security mode, it won't run your add-in, and you can't change the situation simply by obtaining a digital signature, because you can't sign the `mscoree.dll` file.

The solution, then, is to build your own COM component. This is the second time you've done this in this book. If you've been creating tool windows throughout this book, you've been using a "go-between" COM component that loads the .NET custom control that you build with the Visual Studio .NET form designer. This COM control is `VSUserControlHost`, which is *not* a managed .NET assembly; it's just a plain old COM component. If you recall, `VSUserControlHost` is an example of a *shim* component; it's a COM control that acts as a go-between, allowing you to use an assembly as a COM control. Now you can build another shim component, this time one that goes between the Office application and the add-in. But unlike the `mscoree.dll` file, which you cannot digitally sign, this shim component you *can*.



The concept of using a shim component to assist in digitally signing your add-in is not new. Microsoft described the procedure in an article on the MSDN site. To read the article, visit <http://msdn.microsoft.com/library>. In the contents on the left, drill down to Office Solutions Development⇨Microsoft Office⇨Microsoft Office XP⇨Technical Articles⇨Deployment of Managed COM Add-ins in Office XP.

Microsoft has created a control that does the job for you, which you need to download from the MS site. Go to <http://msdn.microsoft.com/code/default.asp>; in the contents on the left, drill down to Code Examples⇨Office Solutions Development⇨Microsoft Office XP⇨Deployment of Managed COM Add-ins in Office XP⇨General Information. From there you can download the component; it's called `odc_shim.exe`.

The idea behind the `odc_shim.exe` control is simple: It does the same job as `mscoree.dll`, except that you get your own private copy of it, which you can digitally sign and distribute with your application. Then, when you install your add-in, instead of pointing the COM server to `mscoree.dll`, you point it to your own shim control.

Because Microsoft has provided a comprehensive step-by-step tutorial on using this shim control, I'm not going to waste space here by rehashing the instructions; I'll simply

point you to the online tutorial: Head over to <http://msdn.microsoft.com/library>. In the contents on the left, expand down to Office Solutions Development→Microsoft Office→Microsoft Office XP→Technical Articles→Using the COM Add-in Shim Solution to Deploy Managed COM Add-ins in Office XP. (If you don't see a contents on the left—as seems to be the case with some versions of Netscape—instead, in the upper-left corner, in the search box, type: COM Add-in Shim Deploy; and in the drop-down box choose MSDN Library. The article should appear first in the search results after the Best Bets section.)

Deploying Macros

Before I get into the details of deploying a macro, I want to emphasize one very important point: When you are ready to deploy a macro, *shut down Visual Studio .NET before you copy the macro files*. The reason is that if you make changes to your macro, it's possible that your changes won't get written to the macro file even though you think they will. The only sure way to have your changes written is by shutting down Visual Studio .NET, allowing it to save all changes.

Throughout this discussion of macro deployment, keep these factors in mind:

- The macros are for Visual Studio .NET, so you can assume the deployment system has all the standard assemblies, as well as the necessary COM components that provide support through the DTE object.
- All the modules for a single macro project live within a single file with a .vsmacros extension.
- Users of the macros have full access to the source code.
- If your macros call into an assembly (many of mine call into VBMacroUtilities), you will want to distribute the assembly with the macros.

If you have a simple set of macros, all in a single project, and the macros do not use any special assemblies besides those that are standard with the Visual Studio .NET and the .NET framework, then deployment is simple: Distribute a copy of your .vsmacros file. The recipients can copy the file into whichever directory they want and then install the file by opening the Macro Explorer (choose View→Other Windows→Macro Explorer), then right-clicking the Macros item in the Macro Explorer, and in the popup menu choosing Load Macro Project, and, finally, browsing to the .vsmacros file and opening it. The file will then be installed.

But there's an even easier way to install the .vsmacros file: Simply *run* it. Have the user copy the file into whichever directory he or she wants and then run the file using one of the many ways, such as double-clicking its name in an Explorer window, browsing to it in the Run dialog box, or typing its name into a DOS prompt. If Visual Studio .NET is installed properly, Windows will then automatically launch the devenv program (the Visual Studio .NET IDE), passing the filename as a parameter; the IDE will install the file, again, automatically. Done deal.

But if you have additional files you want to ship with your macro file, you have some choices to make. You can zip them all up into a single file and let the user unzip

them and copy them into a directory. (Remember, these are users of Visual Studio .NET, hence you can assume, programmers.) Then the users can install the .vsmacros file as before. Or you can get fancy and build a Setup project. I consider a Setup project a mark of professionalism. If somebody gives me a zip file and I create a directory and unzip the files, I'm usually okay with it. But if they give me an installer, I assume they care a great deal about their product and want me to take it seriously.

If you already know how to use the various deployment projects, the following points are for you. And if you aren't familiar with the deployment projects, please read the following points anyway, because afterward, I'll walk you through a setup process:

- If you use the Setup Wizard (rather than the blank Setup project) and add an assembly, the wizard automatically adds the referenced assemblies, such as EnvDTE. You can remove these.
- If you are shipping additional assemblies, make sure you install them into the PublicAssemblies directory.
- Unload the macro project in the Macros Explorer before building the Setup project.

The reason for the first point is that if the users have a valid installation of Visual Studio .NET (which they should, otherwise they wouldn't be downloading macros for Visual Studio .NET), they will already have these additional assemblies. To remove these files from the setup, open the file's properties in the File System Editor and set the Exclude property to True.

As for the second point, you must install the assemblies in the Common7\IDE\PublicAssemblies directory under the main Visual Studio .NET installation directory. Although in the main IDE you can put your assemblies elsewhere and then add a key under HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.0\Assembly-Folders, setting the default to a directory containing your assemblies, unfortunately this technique does not work for the Macros IDE. Therefore, you must put your assemblies in the PublicAssemblies directory.

The reason for the third point is that Visual Studio .NET puts a lock on any macro project files that are loaded. This will cause the Setup project to refuse to build, displaying the following message:

```
Unable to find source file 'C:\dev\Projects\MiscWork\Visual Studio
Projects\VSMacros\OLEProjects\OLEProjects.vsmacros' for file
'OLEProjects.vsmacros', located in '[TARGETDIR]', the file may be absent
or locked.
```

In the sections that follow I show you how to set up the deployment project and an associated project, the custom step project.

Setting Up the Deployment Project

A deployment project (also called a Setup project) describes what the end-user's computer will look like after your software is installed on it. With a Setup project you get

several editors that you can use to describe the end-user's computer. Here are the areas you can configure:

File System. This includes the user's desktop, application directory, and system directory.

Registry. This is the user's system Registry; the editor looks like a typical Registry editor.

File types. This refers to associations between file extensions and a program.

User interface. This includes various dialog boxes for the setup program.

Custom actions. These are scripts or external programs that your setup program will launch to assist in the deployment procedure.

Launch conditions. These are conditions that must exist on the user's computer in order for your setup program to run. The idea here is that you can test for various system requirements before installing your software.

Here are the steps to create a deployment project:

1. Either choose File⇒New⇒Project (if you want to create a new solution) or File⇒Add Project⇒New Project (if you want to use an existing solution). The New Project dialog box will open.
2. In the New Project dialog box, for the Project Types, click Setup and Deployment Projects. For the Templates, click Setup Wizard Project. Type a name and choose a location for your project. Click OK. The Setup Wizard will begin, as shown in Figure 16.4.

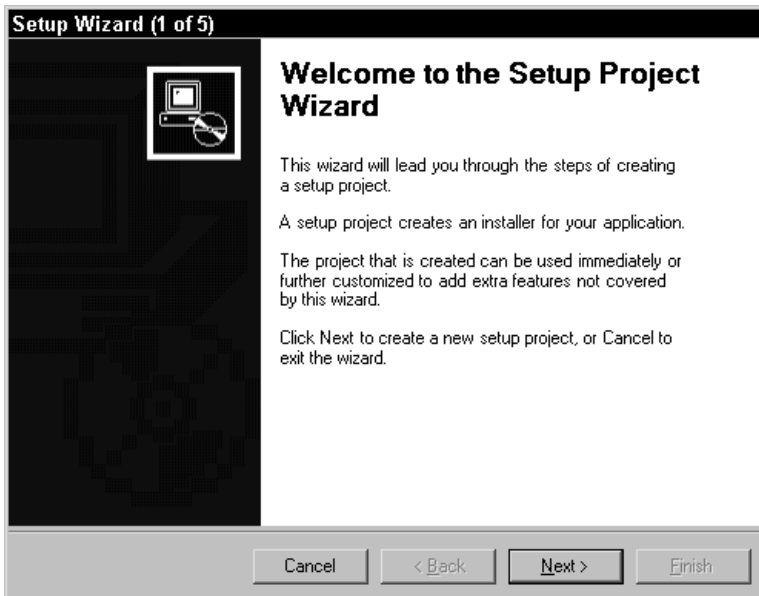


Figure 16.4 The Setup Wizard begins.



The difference between the Setup project and Setup Wizard project is that the Setup Wizard includes a wizard at the beginning that walks you through some steps in filling in a deployment application. The Setup project simply opens with a blank deployment application.

3. On page 1 of the Setup Wizard, the splash screen, click Next.
4. Page 2 of the Setup Wizard, shown in Figure 16.5, asks what type of setup program you want to create. For the macro installer, choose “Create a setup for a Windows application.” Leave the second set of radio buttons, titled “Do you want to create a redistributable package?” blank. (Be careful: as radio buttons, if you click one of them and realize you don’t want either checked, you cannot undo it. You will have to cancel and restart the wizard.) Click Next.
5. Page 3 of the Setup Wizard, shown in Figure 16.6, will appear only if you already have other projects in your solution. Here you can select other projects whose output you want added to the setup. Normally, you would check the project for the program this deployment application is installing. Since you’re installing a macro, not an application, you can leave these blank. Click Next.
6. Page 4 of the Setup Wizard, shown in Figure 16.7, allows you to add files to your deployment project. This is where you choose the assemblies that the macro files require, if any, and the .vsmacros project files. To add a file, click Add; the standard Windows File Open dialog box will open. Browse to your .vsmacros files and click Open; also browse to any assemblies the macro requires, such as my VBMacroUtilities assembly found in the PublicAssemblies directory. Click Next.



Figure 16.5 Page 2 of the Setup Wizard.

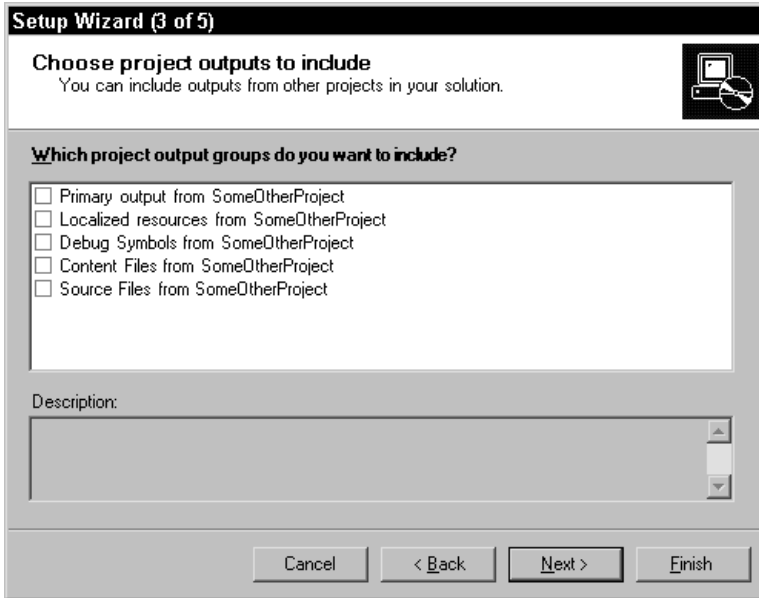


Figure 16.6 Page 3 of the Setup Wizard.

7. Page 5 of the Setup Wizard, shown in Figure 16.8, lists the settings you have chosen. If your settings are the way you want them, click Finish.

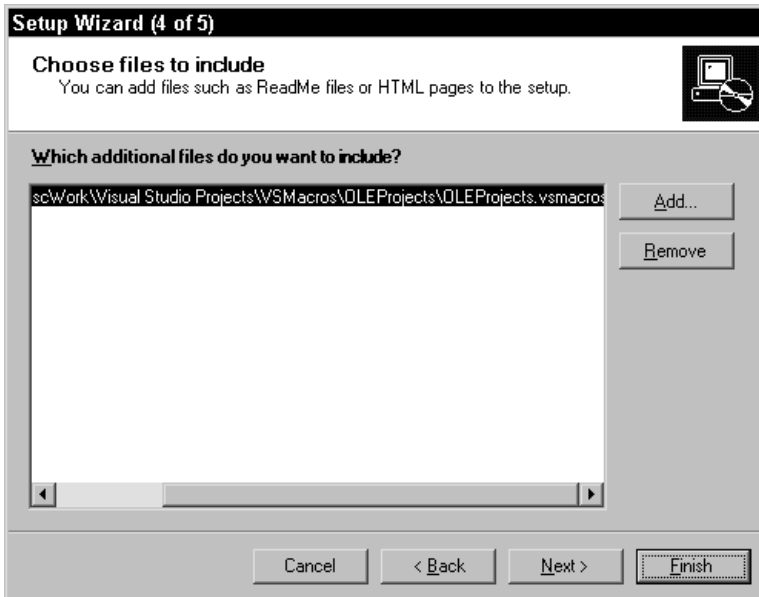


Figure 16.7 Page 4 of the Setup Wizard.

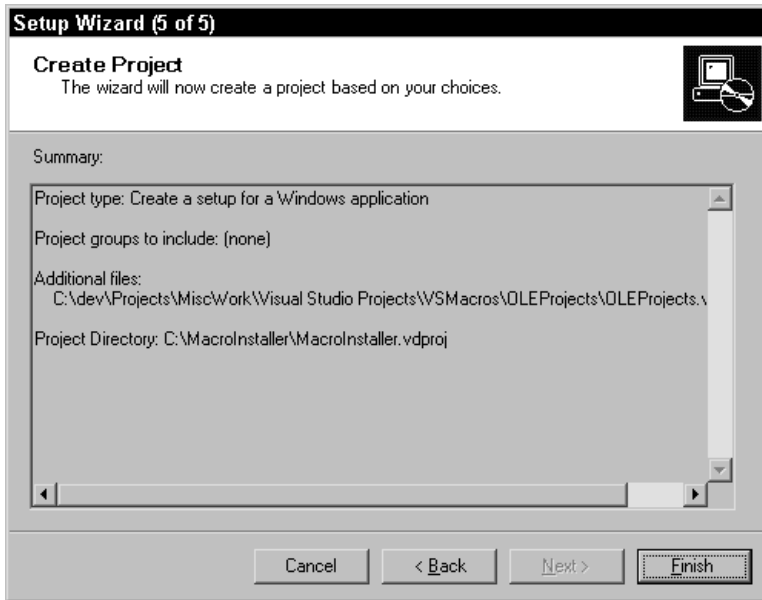


Figure 16.8 Page 5 of the Setup Wizard.

After you click Finish, Visual Studio .NET will create a deployment project for you. At this point, you can simply leave the project as-is, if you want, and you will have a basic installer; all you have to do is build the deployment project. The resulting project will have an .msi extension, which Windows associates with the Windows Installer. Therefore, the end users can simply run the .msi file, and the Windows installer (which is now an official part of Windows, which you can be assured is present) will start up and use the .msi file to install your software.

The deployment project's defaults are pretty good; the final deployment program, when run, will create a directory under the end-user's c:\Program Files directory containing your company name; and under that directory, you'll get a subdirectory with the name of your application.

If you want to change the company name and application name, you can set the properties for the deployment project. To set the properties, single-click the project name in Solution Explorer; then open the Properties Window by choosing View⇨Properties Window. The company name is called Manufacturer in the property window, while the application name is called ProductName.



A deployment project has two property sets. You get one set if you right-click the project in the Solution Explorer, and in the popup menu choose Properties; this opens the window called Property Pages. You can get the second set, called Properties Window, if you single-click the project in the Solution Explorer and then click on the Properties window. It is inside the second set, the Properties Window, where you will find Manufacturer and ProductName.

After you choose your company name and application name, there are two additional actions you can take beyond accepting the defaults. I strongly recommend that you always do the second if you have developed an assembly that you need to ship with your macro. The two actions are:

- Set up the deployment project to create an icon on the user's desktop that installs the .vsmacros file into the user's Visual Studio .NET program. (Remember, the deployment project installs the files on the user's computer. The next step is to install the macro project to Visual Studio .NET, which is what I'm talking about here.)
- Add code to the deployment project to copy the additional assemblies into the PublicAssemblies directory. Unfortunately, the Macros IDE will not look in other directories for its referenced assemblies, even if you add entries to the system Registry.

The first of these involves setting up the File System editor. The second involves creating a Custom Action. In the following two sections I show you how to do both.



To get to the different editors, click on the name of the deployment project in the Solution Explorer. Then click the View menu, where you will find an Editor menu item that otherwise is not present. This Editor menu item has a submenu containing the different deployment editors. Or you can right-click the deployment project in the Solution Explorer, and in the popup menu click View. A submenu will appear listing the editors.

Adding a Shortcut to the .vsmacros File

Probably one of the tasks of your deployment application is to install the .vsmacros file on the user's copy of Visual Studio .NET. When you create a deployment project, your setup can optionally include shortcut icons on the user's desktop. This is handy because you can put a shortcut to the .vsmacros file. The user can then double-click this shortcut, which will launch Visual Studio .NET and automatically install the .vsmacros file.

Remember, the .vsmacros file will be in the subdirectory under Program Files, which your deployment program created on the user's computer. The shortcut on the desktop will be a link to the .vsmacros file. Therefore, after the user installs the .vsmacros file to Visual Studio .NET, he or she is free to delete the shortcut.

The following steps show you how to add a shortcut icon to your application:

1. Open the File System Editor.



You can see the different directories on the user's computer to which you have access by right-clicking the top node on the File System Editor tree on the left, called File System on Target Machine, and in the popup menu choosing Add Special Folder. A submenu will appear listing the different directories.

2. Click Application Folder in the tree on the left. The list of files in the Application Folder will appear in the pane on the right, including any files you added during the wizard process. (Remember, the Application Folder corresponds to the application's directory under the c:\Program Files directory on the user's computer.)
3. Right-click the .vsmacros file in the right pane, and in the popup menu choose Create Shortcut to MyMacros.vsmacros (or whatever your .vsmacros file is called). Optionally, you might type a new name for the shortcut, such as Install MyMacros.vsmacros. Remember, this icon will install the macros into Visual Studio .NET, so make this an appropriate name.
4. The shortcut is now in the Application Folder, but you want it in the User's Desktop folder. To get it there, simply drag it from the right pane to the tree-view on the left over the item called User's Desktop.

That's it. Now when you build the deployment project, the resulting deployment program, when run, will also create an icon on the user's desktop. The user can double-click this icon to install the .vsmacros file into Visual Studio .NET.



When you are developing a deployment application, don't use the Control Panel's Add/Remove Programs setting to uninstall your application. Instead, simply right-click the deployment project in the Solution Explorer, and in the popup menu choose Uninstall. Although the process is the same, it's much faster to do it this way because you don't have to wait for the Add/Remove Program's control panel application to load.

At this point, if you haven't done so already, you can try out your deployment project. Build the project as you would any other project. Then right-click the deployment project and choose Install. This will run the installation program that you just built. After you run the installation, you can uninstall it by right-clicking the project and choosing Uninstall.



After you run your installer, make sure the resulting .vsmacros file is not in use. (It's okay if you have the original source .vsmacros file open.) If you run your installer and your macro is loaded in the IDE, and you then uninstall the macro product, the Windows Installer will complain that the .vsmacros file is in use. To prevent this, make sure you unload the .vsmacros project before uninstalling.

Creating the Custom Action Project

If you want to copy any assemblies to the PublicAssemblies directory, the first thing you will want to do is set up the deployment project so the assemblies will get installed in the application directory. Then you will add a custom step that is a VB.NET program

that copies the files to the PublicAssemblies directory. To determine where the PublicAssemblies directory is, the VB.NET program can use the Registry.

To do this, first create a new Visual Basic .NET Class Library. Make sure you add this project to the existing solution, the one containing your deployment project. To create the Class Library, choose File→Add Project→New Project. In the New Projects dialog box, in the left pane choose Visual Basic Projects. In the right pane choose Class Library. (This is just a .DLL.) Type a name for the DLL, such as MacroExtras.dll. Type a location and click OK.

Now here's the fun part. Many people aren't aware of this next step. In the Solution Explorer, right-click the new MacroExtras project, and in the popup choose Add→Add New Item. In the Templates list in the right pane, scroll down and click Installer Class. Type a name for the class or just leave the default. (I left the default on mine, since the name isn't particularly important here.) Then click Open to create the new source file.

The form designer will open; right-click anywhere on it, and in the popup menu choose View Code. Here's the code you want to type in:

```
Imports System.ComponentModel
Imports System.Configuration.Install
Imports Microsoft.Win32
Imports System.IO

<RunInstaller(True)> Public Class Installer1
    Inherits System.Configuration.Install.Installer

    #Region " Component Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Component Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'Installer overrides dispose to clean up the component list.
    Protected Overrides Sub Dispose(ByVal disposing As
Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Component Designer
    Private components As System.ComponentModel.IContainer
```

```

'NOTE: The following procedure is required by the Component Designer
'It can be modified using the Component Designer.
'Do not modify it using the code editor.
<System.Diagnostics.DebuggerStepThrough()> _
Private Sub InitializeComponent()
    components = New System.ComponentModel.Container()
End Sub

#End Region

Private Sub CopyAssembly(ByVal AssemName As String)
    Try
        Dim reg As RegistryKey
        reg = Registry.LocalMachine.OpenSubKey( _
            "SOFTWARE\Microsoft\VisualStudio\7.0")
        Dim destpath As String = reg.GetValue("InstallDir")
        destpath = Path.Combine(destpath, "PublicAssemblies")
        Dim sourcepath As String = _
            Me.Context.Parameters.Item("InstallPath")
        Dim destfile As String = Path.Combine( _
            destpath, AssemName)
        Dim sourcefile As String = Path.Combine( _
            sourcepath, AssemName)
        File.Copy(sourcefile, destfile)
    Catch
    End Try
End Sub

' Add custom action for Install to make sure this gets called.
' You can click Overrides in upper-left listbox
' and then Install in upper-right to add this header.
Public Overrides Sub Install(ByVal stateSaver As _
System.Collections.IDictionary)
    MyBase.Install(stateSaver)
    Try
        CopyAssembly("VBMacroUtilities.dll")
    Catch e As Exception
        MsgBox("Exception caught: " & e.Message)
    End Try
End Sub

End Class

```

The installer will ultimately call the `Install` function of this code. In this code I call my own `CopyAssembly` function, which reads the Registry and then copies the assembly to the `PublicAssemblies` directory. Since I broke out this functionality into its own subroutine, you can easily add more assemblies if you have them. I also included a `Try/Catch` block for debugging purposes in case anything went wrong.

Now you need to set up the deployment project to run the functions in this library. This involves two steps: adding the project output to your deployment project and setting up the custom action. First, here is how you add the project output:

1. In the Solution Explorer, right-click the deployment project, and in the popup menu choose View⇨File System. In the treeview on the left, right-click Application Folder, and in the popup menu choose Add⇨Project Output. The Add Project Output Group dialog box will appear, as shown in Figure 16.9.
2. Choose the CustomAction1 project in the drop-down box at the top.
3. Click Primary output in the listbox. Set the Configuration set to Active.
4. Click OK.

This adds the project output. Now you can add the custom action. In the Solution Explorer, right-click the deployment project, and in the popup menu choose View⇨Custom Actions. The Custom Actions window will open, as shown in Figure 16.10. (Note that the figure shows what the window will look like after the following steps.)

Now perform these steps:

1. Right-click the word Install in the treeview on the left.
2. In the popup menu choose Add Custom Action. The Select Item in Project dialog box will open.
3. In the Select Item in Project dialog box, double-click Application Folder to switch to the Application folder.
4. Click Primary Output from MacroExtras.dll (active).
5. Click OK.
6. By default, the new item's entry in the treeview will be in edit mode. You can leave the name as-is by clicking anywhere in the tree away from the new item.

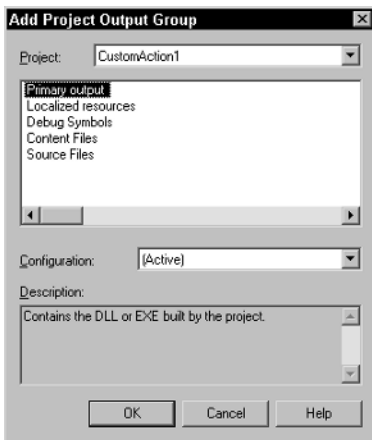


Figure 16.9 The Add Project Output Group dialog box.

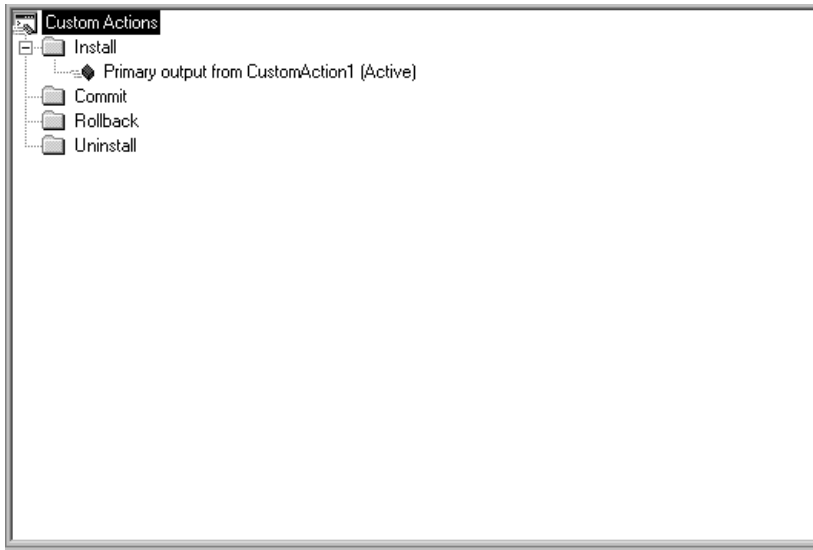


Figure 16.10 The Custom Actions window.

Save your project and you're done. Now when you build the project, the resulting setup will call into the Install subroutine of your MacroExtras.dll.

Here's a code display of the setup at this stage:

```
File System Editor
  Application Folder
    VBMacroUtilities.dll
    Primary Output from CustomAction1 (Active)
  User's Desktop
    Shortcut to OLEProjects.vsmacros
  User's Program Menu
Custom Actions Editor
  Install
    Primary Output from CustomActive1 (Active)
  Commit
  Rollback
  Uninstall
```

As you can see, there's not much there. The deployment project does most of the work for you.

Moving Forward

In this chapter I discussed the security issues you will face when deploying a Microsoft Office add-in to a user's computer. Remember, you do not want to force your users to

lower the security on their application, even if most developers prefer lower security. Many large companies have an IT group that sets up the computers and requires the users to maintain the higher security level.

I also showed you how you can put together a Setup project that will deploy your macros for you. If you're really adventurous, you might want to take another step and create a wizard that builds Setup projects specifically for macros. To do so, use the techniques for creating a wizard that I described in Chapter 12. And if you'd like to share it with me, send it to readers@jeffcogswell.com; if it works, I'll upload it to the Web site for this book.

Supercharging Visual Studio .NET

In this, the final chapter of the book, I wrap-up add-in development in Visual Studio .NET by, first, showing you how you can create a new page in the Options dialog box through which the IDE users can configure your add-in; next, providing another add-in I've found a need for and thought you might, too; and then listing a few handy third-party add-ins that you might want to obtain.

I close the chapter and the book by talking a bit about the Common Language Infrastructure (CLI) and where to go if you want to take your development further. This final section doesn't mention add-ins and macros, but when your mind-set is on enhancing Visual Studio .NET, you'll find that a whole world exists beyond them. For example, you can enhance your .NET programming by buying another language add-in, such as Python or Perl, and I'll tell you where you can find such products.

Creating an Options Page for Your Add-in

If the add-in you're creating has various options that the IDE user can configure, a good place to put the options is directly in the main IDE's Options dialog box, which is available by choosing Tools⇨Options. Creating an options page involves writing an ActiveX control that implements the `IDTToolsOptionsPage` interface and registering the options page in the Visual Studio .NET section of the Registry.

Keep in mind that when I say that your control must implement the `IDTToolsOptionsPage`, I mean that your control must contain a class derived from it, overriding the member functions of `IDTToolsOptionsPage`. These members are:

- GetProperties.** The options page can return a set of properties, although it doesn't have to. (But you still must implement this function, even if you return a null value.)
- OnAfterCreated.** This event occurs the first time the options page is loaded during the Visual Studio .NET session. (The next time the IDE user restarts Visual Studio .NET and opens the options page, this event will again occur.)
- OnCancel.** This event occurs when the IDE user clicks the Cancel button in the Options dialog box.
- OnHelp.** This event occurs when the IDE user has your options page open and clicks the Help button.
- OnOK.** This event occurs when the user clicks the OK button in the Options dialog box.

In Chapter 7, "Creating Add-ins for the IDE," in the section "Using the Form Designer with a Tool Window," I introduced the idea of a shim control, a control that provides the necessary COM portions for a .NET control so that the .NET control can perform where an ActiveX control is required. In order to create an options page, you must provide Visual Studio .NET with an ActiveX control. Since the .NET assemblies do not serve as COM components (and therefore do not serve as ActiveX controls, either), you must either develop a non-.NET ActiveX control using C++ (or resort to an earlier version of Visual Basic) or you must use a shim control, which allows you to do the rest of your development in Visual Studio .NET using VB.NET or C#. The latter option is the approach I take here.



The shim control used in the first sample here is available free for download from the Microsoft Web site. The control was originally included in an article in *MSDN Magazine* titled "Custom Add-ins Help You Maximize the Productivity of Visual Studio .NET" (February 2002). You can find both the article and the control at <http://msdn.microsoft.com/msdnmag/issues/02/02/VSIDE/default.aspx>. The shim control is the *VSIDE.exe* file at the top; it's a self-extracting executable that also contains a sample project. (Also note that *MSDN Magazine* works closely with Microsoft, and you can download all of the publication's source code from the [msdn.Microsoft.com](http://msdn.microsoft.com) site.)

Once you have obtained the shim control, go ahead and build the sample solution that accompanies it. Doing so will both build the shim control and register it with the operating system, as well as allow you to poke around the sample add-in that accompanies the control. Next start a new solution and create a new add-in project. Create this project in VB.NET and select the Tools Menu option so that your add-in will implement the `IDTCommandTarget` interface.

In order to focus on the options page and not get bogged down in the add-in itself, I decided to make this a simple, yet useful add-in, one that does not involve a tool window; rather, it simply scans through all the documents in the current project and prints document statistics to the output window, specifically character count and line count.

The shim control for options pages work a little differently from the shim control for tool windows. With the add-in projects that use the tool windows, you need to add a reference to the tool window shim control; here you do not. Instead, you reference the shim control through the code using a GUID. As you'll see in the following sample code, in the code for the user control, you will provide a GuidAttribute like so:

```
<GuidAttribute("55B20E98-768F-4d16-BAEB-B613B49653B9"), _
ProgId("StatsAddin.OptionsControl")>
```

Or, if your code is C#, the same code will look like this:

```
[GuidAttribute("55B20E98-768F-4d16-BAEB-B613B49653B9"),
ProgId("StatsAddin.OptionsControl")]
```

Note that you must use the GUID that I'm providing here, which matches the GUID in the shim control; that's how you reference the shim control in your code. For the ProgId, however, you provide a new name for each add-in. The name I'm giving here, StatsAddin, goes with my sample statistics add-in. (I recommend following the name by a dot and the word OptionsControl, for consistency with other add-ins that use this shim control.)

And now on to the add-in code itself, which is straightforward:

```
Imports Microsoft.Office.Core
imports Extensibility
imports System.Runtime.InteropServices
Imports EnvDTE

<GuidAttribute("27ABE2EA-CB9A-4F2D-A046-6C6EA5AC5C64"), _
ProgIdAttribute("StatsAddin.Connect")> _
Public Class Connect

    Implements Extensibility.IDTExtensibility2
    Implements IDTCommandTarget

    Dim applicationObject As EnvDTE.DTE
    Dim addInInstance As EnvDTE.AddIn
    Dim pane As OutputWindowPane

    Public Sub OnBeginShutdown(ByRef custom As System.Array) _
        Implements Extensibility.IDTExtensibility2.OnBeginShutdown
    End Sub

    Public Sub OnAddInsUpdate(ByRef custom As System.Array) _
        Implements Extensibility.IDTExtensibility2.OnAddInsUpdate
    End Sub

    Public Sub OnStartupComplete(ByRef custom As System.Array) _
        Implements Extensibility.IDTExtensibility2.OnStartupComplete
    End Sub
```

```

Public Sub OnDisconnection(ByVal RemoveMode As _
    Extensibility.ext_DisconnectMode, ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnDisconnection
End Sub

Public Sub OnConnection(ByVal application As Object, _
    ByVal connectMode As Extensibility.ext_ConnectMode, _
    ByVal addInInst As Object, ByRef custom As System.Array) _
    Implements Extensibility.IDTExtensibility2.OnConnection

    applicationObject = CType(application, EnvDTE.DTE)
    addInInstance = CType(addInInst, EnvDTE.AddIn)
    If connectMode = Extensibility.ext_ConnectMode.ext_cm_UISetup
Then
        Dim objAddIn As AddIn = CType(addInInst, AddIn)
        Dim CommandObj As Command
        Try
            CommandObj = applicationObject.Commands.AddNamedCommand _
                (objAddIn, "StatsAddin", "StatsAddin", _
                "Executes the command for StatsAddin", True, 59, _
                Nothing, 1 + 2)
            CommandObj.AddControl(applicationObject.CommandBars. _
                Item("Tools"))
        Catch e As System.Exception
        End Try
    End If

    ' Create the output pane
    Dim outwin As Window = applicationObject.Windows.Item( _
        EnvDTE.Constants.vsWindowKindOutput)
    outwin.Visible = True
    pane = outwin.Object.OutputWindowPanes.Add("Statistics")
    WriteAllStats()

End Sub

Public Sub Exec(ByVal cmdName As String, _
    ByVal executeOption As vsCommandExecOption, _
    ByRef varIn As Object, ByRef varOut As Object, _
    ByRef handled As Boolean) Implements IDTCommandTarget.Exec
    handled = False
    If (executeOption = _
        vsCommandExecOption.vsCommandExecOptionDoDefault) Then
        If cmdName = "StatsAddin.Connect.StatsAddin" Then
            WriteAllStats()
            handled = True
        End If
    End If
End Sub

```

```

        End If
    End If
End Sub

Public Sub QueryStatus(ByVal cmdName As String, _
    ByVal neededText As vsCommandStatusTextWanted, _
    ByRef statusOption As vsCommandStatus, ByRef commandText As Object) _
    Implements IDTCommandTarget.QueryStatus
    If neededText = EnvDTE.vsCommandStatusTextWanted. _
        vsCommandStatusTextWantedNone Then
        If cmdName = "StatsAddin.Connect.StatsAddin" Then
            statusOption = CType(vsCommandStatus. _
                vsCommandStatusEnabled + vsCommandStatus. _
                vsCommandStatusSupported, vsCommandStatus)
        Else
            statusOption = vsCommandStatus. _
                vsCommandStatusUnsupported
        End If
    End If
End Sub

```

```

Public Sub WriteAllStats()
    pane.OutputString("Statistics" & Chr(13))
    Dim proj As Project
    For Each proj In applicationObject.ActiveSolutionProjects
        pane.OutputString(proj.Name & Chr(13))
        WriteStats(proj)
    Next
End Sub

```

```

Public Sub WriteStats(ByVal proj As Project)
    Dim DoLineCount As Integer
    Dim DoCharCount As Integer
    Dim key As Microsoft.Win32.RegistryKey
    key = Microsoft.Win32.Registry.LocalMachine.OpenSubKey( _
        "Software\Microsoft\VisualStudio\7.0\" & _
        "Addins\StatsAddin.Connect\Options")

    DoLineCount = key.GetValue("LineCount", 1)
    DoCharCount = key.GetValue("CharCount", 1)
    Dim pitem As ProjectItem
    For Each pitem In proj.ProjectItems
        Dim filename As String
        Dim filenum As Integer
        For filenum = 1 To pitem.FileCount
            filename = pitem.FileNames(filenum)
            Dim ext As String
            ext = System.IO.Path.GetExtension(filename)
            If ext = ".vb" Or ext = ".cpp" Or ext = ".cs" Then
                Dim shortname As String

```



```

        Dim reader As System.IO.StreamReader
        shortname = System.IO.Path.GetFileName(filename)
        reader = System.IO.File.OpenText(filename)
        pane.OutputString("    " & shortname & " ")
        If DoCharCount = 1 Then
            pane.OutputString(reader.BaseStream.Length & " ")
        End If
        If DoLineCount = 1 Then
            pane.OutputString(LineCount(reader))
        End If
        pane.OutputString(Chr(13))
    End If
Next
Next
End Sub

Private Function LineCount( _
ByRef reader As System.IO.StreamReader) As Integer
    Dim line As String
    line = reader.ReadLine()
    If line = "" Then
        Return 0
    End If
    Dim count As Integer = 1
    While (reader.Peek() > -1)
        line = reader.ReadLine()
        count += 1
    End While
    Return count
End Function

End Class

```

When you add the code to save the changes in the form, you may be tempted to save them to local variables that the `Connect` module can access. However, that's not the correct way to write the code, since you want the settings to be available the next time the IDE user starts the IDE, even if he or she doesn't open the options page during this second session. The correct approach, then, is to save the changes to the Registry under your add-in entry. Your `Connect` module then reads the entries in the Registry, not the data stored in the form.

Figure 17.1 shows the layout for the form, which includes two checkboxes. To create this form, add a new user control to the project (not a new form). To do this, right-click the project in the Solution Explorer, and in the popup menu choose `Add` → `Add User Control`. In the `Add New Item` dialog box, I chose to call the control filename `StatsControl.vb`. Set the top checkbox's name to `CharCountCheck` and its `Text` property to `Character Count`. Set the bottom checkbox's name to `LineCountCheck`, and its `Text` property to `Line Count`.

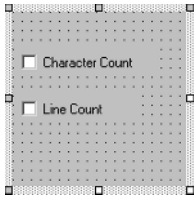


Figure 17.1 The layout for the form includes two checkboxes.

Following is the code for the form. Notice that I've changed the header for the class by adding the `GuidAttribute` section that I mentioned earlier; I also added a class, `IDTToolsOptionsPage`, to the base class list.

```
Imports System.Runtime.InteropServices ' For GuidAttribute

<GuidAttribute("55B20E98-768F-4d16-BAEB-B613B49653B9"), _
ProgId("StatsAddin.OptionsControl")> _
Public Class StatsControl
    Inherits System.Windows.Forms.UserControl
    Implements EnvDTE.IDTToolsOptionsPage

#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'UserControl overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As
Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.IContainer
```

```
'NOTE: The following procedure is required by the Windows Form
'Designer
'It can be modified using the Windows Form Designer.
'Do not modify it using the code editor.
Friend WithEvents CharCountCheck As System.Windows.Forms.CheckBox
Friend WithEvents LineCountCheck As System.Windows.Forms.CheckBox
<System.Diagnostics.DebuggerStepThrough()> _
Private Sub InitializeComponent()
    Me.CharCountCheck = New System.Windows.Forms.CheckBox()
    Me.LineCountCheck = New System.Windows.Forms.CheckBox()
    Me.SuspendLayout()
    '
    'CharCountCheck
    '
    Me.CharCountCheck.Location = New System.Drawing.Point(8, 32)
    Me.CharCountCheck.Name = "CharCountCheck"
    Me.CharCountCheck.Size = New System.Drawing.Size(120, 24)
    Me.CharCountCheck.TabIndex = 0
    Me.CharCountCheck.Text = "Character Count"
    '
    'LineCountCheck
    '
    Me.LineCountCheck.Location = New System.Drawing.Point(8, 72)
    Me.LineCountCheck.Name = "LineCountCheck"
    Me.LineCountCheck.TabIndex = 1
    Me.LineCountCheck.Text = "Line Count"
    '
    'StatsControl
    '
    Me.Controls.AddRange(New System.Windows.Forms.Control() { _
        Me.LineCountCheck, Me.CharCountCheck})
    Me.Name = "StatsControl"
    Me.ResumeLayout(False)

End Sub

#End Region

' Note: This function's param is ByRef not ByVal
Public Sub GetProperties(ByRef PropertiesObject As Object) _
    Implements EnvDTE.IDTTToolsOptionsPage.GetProperties
    PropertiesObject = Nothing
End Sub

Public Sub OnAfterCreated(ByVal DTEObject As EnvDTE.DTE) _
    Implements EnvDTE.IDTTToolsOptionsPage.OnAfterCreated
    ' Read the entries from the registry
    ' and fill in the form based on the entries
    Dim DoLineCount As Integer
    Dim DoCharCount As Integer
```

```

Dim key As Microsoft.Win32.RegistryKey
key = Microsoft.Win32.Registry.LocalMachine.OpenSubKey( _
    "Software\Microsoft\VisualStudio\7.0\" & _
    "Addins\StatsAddin.Connect\Options")

DoLineCount = key.GetValue("LineCount", 1)
DoCharCount = key.GetValue("CharCount", 1)
If DoLineCount = 1 Then
    LineCountCheck.Checked = True
Else
    LineCountCheck.Checked = False
End If
If DoCharCount = 1 Then
    CharCountCheck.Checked = True
Else
    CharCountCheck.Checked = False
End If
End Sub

Public Sub OnCancel() _
Implements EnvDTE.IDTToolsOptionsPage.OnCancel
    ' Don't save the entries, since the user
    ' clicked Cancel.
End Sub

Public Sub OnHelp() _
Implements EnvDTE.IDTToolsOptionsPage.OnHelp
End Sub

Public Sub OnOK() _
Implements EnvDTE.IDTToolsOptionsPage.OnOK
    ' Save the entries into the registry
    Dim DoLineCount As Integer
    Dim DoCharCount As Integer
    Dim key As Microsoft.Win32.RegistryKey
    ' Remember to pass True for the second
    ' parameter to OpenSubKey, which will
    ' allow write access to the key.
    key = Microsoft.Win32.Registry.LocalMachine.OpenSubKey( _
        "Software\Microsoft\VisualStudio\7.0\" & _
        "Addins\StatsAddin.Connect\Options", True)
    If LineCountCheck.Checked Then
        DoLineCount = 1
    Else
        DoLineCount = 0
    End If
    If CharCountCheck.Checked Then
        DoCharCount = 1
    
```