



Visual Studio Add-Ins

Succinctly

by Joe Booth

Visual Studio Add-Ins Succinctly

By
Joe Booth

Foreword by Daniel Jebaraj



Copyright © 2013 by Syncfusion Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: Jay Natarajan, senior product manager, Syncfusion, Inc.

Copy Editor: Courtney Wright

Acquisitions Coordinator: Jessica Rightmer, senior marketing strategist, Syncfusion, Inc.

Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	13
About the Author	15
Preface	16
Chapter 1 Microsoft Visual Studio	18
Visual Studio add-ins	18
IDTExtensibility2 Interface	18
IDTCommandTarget Interface	19
Assemblies	19
Wizard	19
Chapter 2 Add-in “Hello World”	20
Create the project	20
Select your language	21
Application hosts	21
Name and Description	21
Add-in options	22
About Information	23
Summary	23
Connection Code	23
Exec Code	25
Query Status code	25
Generated files	26
Chapter 3 Hooking into the IDE	27
OnConnection Method	27

Linking to menu items.....	27
Linking to other windows.....	29
Adding to the code window.....	29
Other IDE windows	30
Adding a toolbar button.....	31
QueryStatus Method	31
Other methods	32
OnAddInsUpdate method	32
OnBeginShutdown method.....	32
OnDisconnection method	32
On StartupComplete	32
A few caveats.....	32
Chapter 4 Application and Add-in Objects.....	34
Application Object	34
ActiveDocument.....	34
ActiveWindow	34
Debugger	34
Documents.....	34
Edition	35
ItemOperations	35
LocaleID.....	35
MainWindow	35
Mode	35
Solution	35
ToolWindows	35
Windows	36
AddIn Object	36

Add-in properties	36
Collection	36
Connected	36
Description	36
GUID	37
Name	37
Object.....	37
ProgID.....	37
SatelliteDLLPath	37
Assemblies.....	37
Extensibility.dll	37
CommandBars.dll	37
EnvDTE.dll	38
VSLangProj.dll	38
Chapter 5 Save Some Files Add-In.....	39
SaveSomeFiles add-in.....	39
Designing the selection form.....	39
Implementing the Exec() method.....	40
But not while debugging.....	41
Summary.....	42
Chapter 6 Testing Your Add-In	43
Configuration files	43
For Testing.AddIn	43
Add-in settings	43
LoadBehavior.....	43
CommandPreload	44
Add-in life cycle	44

0: Call Manually	44
1: Load at Start-up	45
Debugging.....	45
Common mistakes	45
Add-in not enabled on menu	45
Add-in never invoked	46
Events not triggering	46
Not seeing code changes	46
Removing an add-in module	46
Pesky “Unable to delete” message	46
Chapter 7 Visual Studio Environment.....	47
VS Info Wizard	47
VS Info Form	47
Exec() method.....	48
Getting options.....	49
Getting add-ins installed	51
Environment information	51
Getting an OS-friendly name	52
Displaying the form	53
Final results.....	53
Chapter 8 Solution	55
Solution Info Wizard	55
Updating the menu icon	55
Exec() method.....	56
Solution info	56
Totaling project information	56
Properties	58

Displaying the results	58
Solution methods	59
Close	59
FindProjectItem	59
SaveAs	59
SolutionBuild	60
Build	60
Clean	60
Run	60
BuildState	60
Chapter 9 Projects	61
Project Info Wizard	61
Exec() method	61
Getting each project	62
Project type	62
VSProject type	63
References	63
Project Items	63
ITEMS	64
Adding the JavaScript	65
Showing the Results	65
Styling the HTML	65
Chapter 10 IDE Windows	67
Windows	67
Tool windows	67
Document windows	67
Window object	68

Properties	68
Methods	68
ActiveWindow	68
MainWindow	68
Windows.....	69
Window Kind constants	69
Tool windows	70
Document windows.....	70
Is AJAX being used?.....	71
Getting the active window.....	72
Making sure it is HTML code	72
Parsing the HTML code	72
Showing our findings	73
Summary.....	74
Chapter 11 Documents	75
Getting the document.....	75
Document object.....	75
Text document object	76
Converting C# to VB	77
Summary.....	77
Chapter 12 Code Window.....	78
Simple code manipulation.....	78
Attaching to the code window.....	78
Responding to the click.....	79
Getting selected code	79
Tweak the code fragment	80
Putting the code back	80

Moving the code around	81
Text Document	81
Edit point	82
More complex code manipulation	83
Chapter 13 Code Model	84
Using the code model	84
Get the code model of a source file	85
Code element properties	85
Putting it all together	86
Class documenter	88
Attaching to the code editor window	88
Getting the code model	89
Finding the class elements	90
Building our header	91
Organizing the code elements	92
Variables	93
Enums	93
Properties	94
Methods	94
Writing the header back to the source window	95
Summary	95
Chapter 14 Tool Windows	96
Error List	96
Task List	97
Solution Explorer	98
Output Window	98
Searching for bad words	99

Bad words scan	99
Using a tool button.....	100
Only if a solution is open.....	100
Getting tool windows.....	101
Looping through the project.....	101
Marking bad words	101
Adding a clean-up task	102
Summary.....	102
Chapter 15 Source Code Generation	103
Source code helper class.....	103
Standardized headers	106
Wizard settings	106
Moving to File menu.....	107
Options screen.....	107
Generate the header	108
Add sub/function call.....	109
Add standard variables	109
Open a new window.....	109
Item Operations object.....	110
Summary.....	110
Chapter 16 Deploying Your Add-In	111
Installing the add-in.....	111
Add-in Manager	112
Summary.....	112
Chapter 17 Object Reference	113
Application Object (DTE2)	113
Windows and documents.....	114

Document	114
Window	115
Solution and projects	115
Solution	116
Project	116
Project Item	117
Code manipulation	118
Text Document	118
Edit Point	118
Code Model	119
Code Element	120
Chapter 18 Add-in Helper Class	121
MakeEmptySolution	121
GetVSProjectsFolder	121
FindMenuIndex	122
Chapter 19 Third-Party Add-Ins	123
Microsoft add-ins	123
Community add-ins	123
Indent Guides	123

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Joseph D. Booth has been programming since 1981 in a variety of languages, including BASIC, Clipper, FoxPro, Delphi, Classic ASP, Visual Basic, and Visual C#. He has also worked in various database platforms, including DBASE, Paradox, Oracle, and SQL-Server from version 6.5 up through SQL 2012.

He is the author of six computer books on Clipper and FoxPro programming, Network Programming, and Client/Server development with Delphi. He also wrote several third-party developer tools, including CLIPWKS, which allowed the ability to programmatically create and read native Lotus and Excel spreadsheet files from Clipper applications.

Joe has worked for a number of companies including Sperry Univac, MCI-WorldCom, Ronin, Harris Interactive, Thomas Jefferson University, People Metrics, and Investor Force. He is one of the primary authors of Results for Research (market research software), *PEPSys* (industrial distribution software) and a key contributor to *AccuBuild* (accounting software for the construction industry).

He has a background in accounting as well, having worked as a controller for several years in the industrial distribution field, although his real passion is computer programming.

In his spare time, Joe is an avid tennis player and a crazy soccer player (he plays goalie). He also practices yoga and martial arts (holding a brown belt in Judo).

Preface

Target Audience

This book is for developers who are currently using Microsoft Visual Studio and want to add their own custom features to that development environment. It assumes you are comfortable programming in C# and are also comfortable writing classes and class methods to implement interfaces. It is designed to provide a quick overview of how to create an add-in, how to test your add-in, and how to install and share it. There are a number of add-in modules to provide working examples to whet your appetite.

The focus of this book is the add-in ability in Visual Studio; it does not cover the more powerful, but substantially more complex, package add-in feature of Visual Studio.

Tools Needed

In order to be able to follow along with all of the examples in this book, you will need Microsoft Visual Studio 2010 or Visual Studio 2012.

Many of the examples may work in older versions of Visual Studio as well. The extensibility features have been in the IDE since Visual Studio release in 1997. Note, however, that add-in modules are not supported in Express editions of Visual Studio.

Formatting

Throughout the book, I have used several formatting conventions.



Note: *Ideas and notes about the current topic.*



Tip: *Ideas, tips, and suggestions.*

Code Blocks

```
public void Exec(string commandName, vsCommandExecOption executeOption, ref object varIn,
ref object varOut, ref bool handled)
{
    handled = false;
}
```

Using Code Examples

All code samples in this book are available at https://bitbucket.org/syncfusion/visualstudio-add-ins_succinctly/.

Chapter 1 Microsoft Visual Studio

Microsoft's Visual Studio is one of the most popular integrated development environments (IDE) available today. Yet as popular and powerful as Visual Studio is, there may be times when you want to add your own quirks to the tool. And fortunately, Microsoft makes it pretty easy to do just that. You can create add-ins to the various menus and toolbars to perform a variety of tasks, pretty much anything you can program. The add-in can be written in Visual Basic, C#, or C++; there are no arcane or additional languages to learn.

Visual Studio has been around in various incarnations since the late 1990s. Early Microsoft IDE products were separate for the language you were working in; Visual Basic was one tool, Visual C++ another, etc. However, with the release of Visual Studio 97, Microsoft began to bundle the various programming languages into the same IDE. Visual Studio 97 included Visual Basic, Visual C++, Visual J++, Visual FoxPro, and Visual Interdev.

When Microsoft created Visual Studio 97, it was built as an extensible core platform, making it easier for Microsoft developers to integrate new features into the IDE. They also allowed outside developers to write add-ins to enhance the product using the same extensible platform that the Visual Studio engineers worked in.

As the Visual Studio platform continued to grow, third-party developers continually wrote add-ins to integrate tools into Visual Studio. Shortly after the release of Visual Studio 2008, Microsoft created a website called the [Visual Studio Gallery](#). New tools and enhancements are added, and as of this writing, there are more than 3,000 add-ins listed in the gallery.

The extensibility built into Visual Studio makes it an excellent environment to start and build your own “improvements” to the IDE. Getting started down that path is what this book is all about.

Visual Studio add-ins

To build a Visual Studio add-in, you will need to create a new class that will provide implementation methods for two interfaces from the **Extensibility** and **EnvDTE** namespaces. An interface is a module containing declarations of methods and events, but with no implementation provided. This approach allows your add-in to plug and play into the Visual Studio IDE.

You will also need to generate an XML configuration file, which tells Visual Studio how to load your add-in and where your add-in's assembly code file (DLL) can be found.

IDTExtensibility2 Interface

This interface from the Extensibility namespace is used to hook your add-in into the Visual Studio IDE. Although you will need to create method implementations for each of the interface events, only the **OnConnection** method is needed to get your add-in loaded into the IDE.

IDTCommandTarget Interface

This interface from the EnvDTE namespace is used to process a request from the IDE to run your add-in. The first parameter to both methods is the command name, so your add-in code knows which (of possibly multiple) commands Visual Studio is asking about.

Assemblies

When implementing an add-in module, the following assemblies need to be included in the project:

- Extensibility
- EnvDTE

There are later versions of the EnvDTE assembly, which add on additional classes, enums, and interfaces. If you choose to implement an add-in that interacts with some of the later versions of Visual Studio, you may need to include these assemblies in your project as well:

- EnvDTE: All versions of Visual Studio.
- EnvDTE80: VS 2005 and above, interfaces typically ending with 2, e.g., EnvDTE2.
- EnvDTE90: VS 2008 and above, interfaces ending with 3, e.g., HTMLWindow3.
- EnvDTE100: VS 2010 and above.

When you create an add-in module using the Add-in Wizard, EnvDTE and EnvDTE80 are typically included for you.

Wizard

Visual Studio's **New Project** menu includes a wizard that will generate most of the code you need to integrate your add-in into the IDE. It will also generate the XML configuration file to allow the IDE to find and load your add-in program. In the next chapter, we will use this wizard to create the famous "Hello World" programming example.

Chapter 2 Add-in “Hello World”

Ever since the classic example in the book *The C Programming Language*, the Hello World program has been the starting point for new example programs. In this chapter, we will use the project wizard to create a Visual Studio add-in version of this classic example.

Create the project

To create a new add-in project, we will use the Add-in Wizard built into Visual Studio:

1. Open Visual Studio and select **New Project** on the **File** menu.
2. Choose **Other Project Types** from the **Installed Templates** list.
3. Choose **Extensibility**.

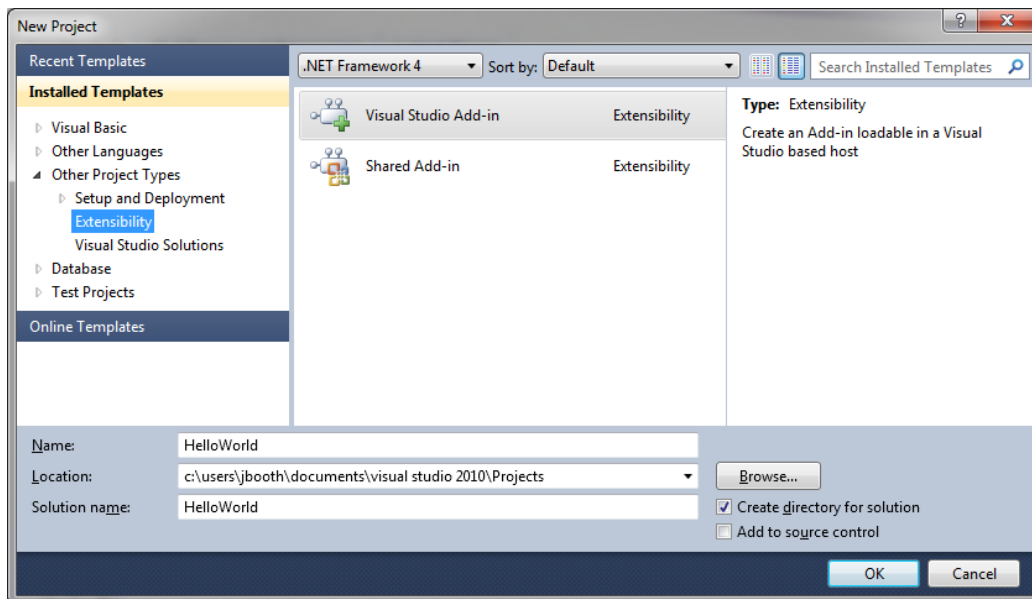


Figure 1: Creating a new project

There are two types of add-ins you can create, one that can be loaded into Visual Studio (which is the focus of this book), as well as a shared add-in that can be used across different Microsoft products (such as Word, Excel, Outlook, etc.).



Note: All of the add-in modules we create in this book will start with the wizard screen, so you will use it quite a bit. It is definitely a time-saver compared to creating the implementation class code and XML files manually.

Select your language

After the wizard splash screen, you will be given the option to select the programming language you want the code to be generated in. The options are:

- Visual C#
- Visual Basic
- Visual C++ /CLR
- Visual C++/ATL

For the examples in this book, we will work in Visual C#, but you may use whichever language you are most comfortable programming in. This choice determines the language in which the add-in project will be generated, but does not impact running or using the add-in.

Application hosts

The application hosts selection screen lets you indicate which host applications can run your add-in. The options are:

- Visual Studio
- Visual Studio Macros

You can select either or both options. For the examples in this book, we only need to select Visual Studio. The add-in XML file will contain a `<HostApplication>` entry for each option selected. Most add-ins in this book will have a UI component, so you shouldn't need to select Visual Studio Macros.



Note: When using Visual Studio macros, interactive commands such as *LaunchWizard*, *OpenFile*, etc. are not available. Other commands, such as *Quit*, behave differently. For example, the *Quit* command closes the Macros IDE and returns to Visual Studio, rather than shutting down Visual Studio itself.

Name and Description

You can provide a name and description for your add-in. The name will be used as the menu label as well as the internal name for the command your code implements. The description is stored as tooltip text, which the IDE displays when the user selects the add-in from the **Add-in Manager** window.

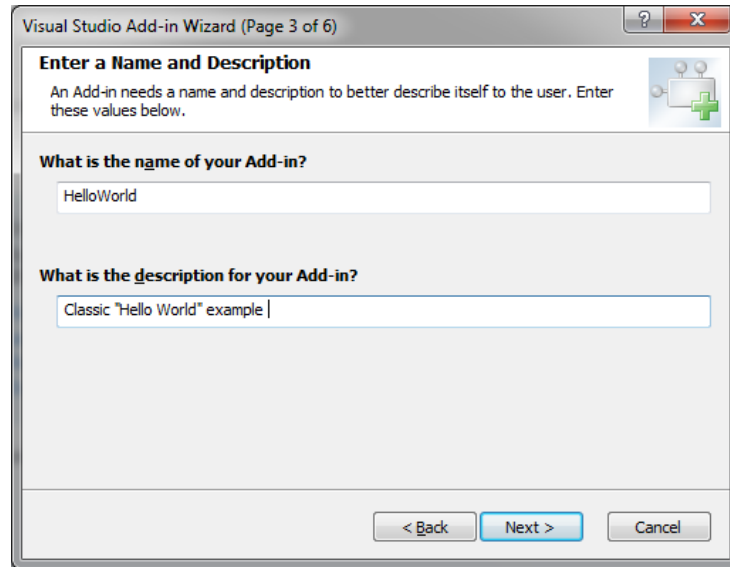


Figure 2: Adding a name and description to the add-in



Note: The wizard will generate a unique, qualified command name consisting of `<filename>.Connect.<commandName>` when referencing your add-in module's commands.

Add-in options

The add-in options screen helps control the generated code for your add-in. In our examples, we are going to hook our class into the **Tools** menu, so we select the first option to generate code on the Connection method to load our add-in.

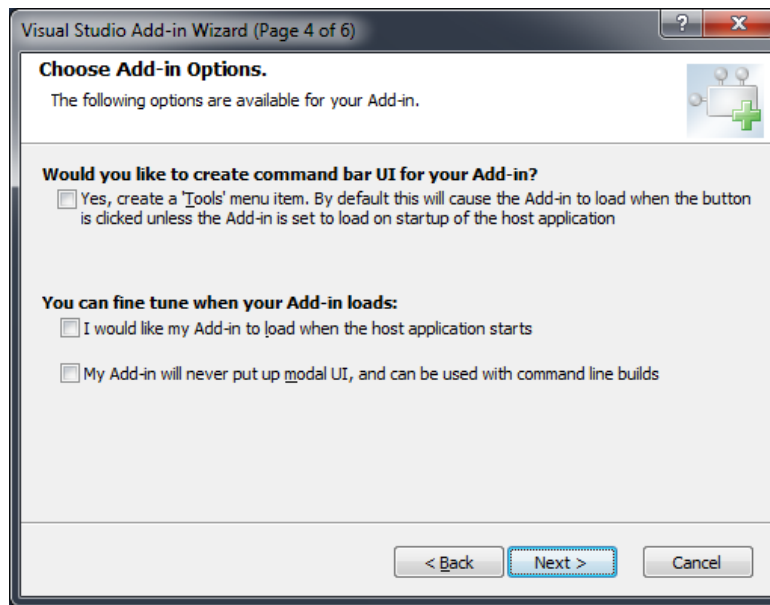


Figure 3: Add-in options

For debugging purposes, do not select the **Load Add-in** check box when the host application starts. Ignoring this option will make debugging easier. When you are ready to deploy your application, it is an easy update to have your add-in load at start-up time.



Note: In some examples, we might connect to a different menu or toolbar, but it is still beneficial to let the wizard generate the default method, even if we tweak its code.

About Information

The **About Information** option lets you specify text to display in the Visual Studio **About** box. When you run the **About Visual Studio** menu option, the dialog box displays a list of all installed products, including add-ins. When a user navigates to your add-in, any information you provide will be displayed below the add-in list.

Summary

After you have filled in all of the information, a summary screen will be shown:

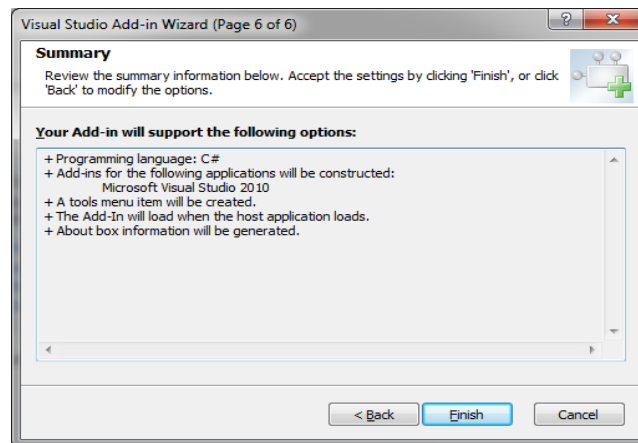


Figure 4: Summary of add-in options

Double-check your selections, and if they all look okay, click **Finish**. The wizard will work for a bit, and then produce a source file that provides implementation methods for the **IDTExtensibility2** and **IDTCommandTarget** interfaces. It will also generate the XML configuration files with your add-in load instructions.

Connection Code

The code generated by the **OnConnection** method will look like the following code sample (it may be different depending upon language and settings):

```

public void OnConnection(object application, ext_ConnectMode connectMode,
                        object addInInst, ref Array custom)
{
    _applicationObject = (DTE2)application;
    _addInInstance = (AddIn)addInInst;
    if(connectMode == ext_ConnectMode.ext_cm_UISetup)
    {
        object []contextGUIDS = new object[] { };
        Commands2 commands = (Commands2)_applicationObject.Commands;
        string toolsMenuName = "Tools";
        //Place the command on the tools menu.
        //Find the MenuBar command bar,
        CommandBars.CommandBar menuBarCommandBar =
        ((CommandBars.CommandBars)_applicationObject.CommandBars)["MenuBar"];
        //Find the Tools command bar on the MenuBar command bar:
        CommandBarControl toolsControl = menuBarCommandBar.Controls[toolsMenuName];
        CommandBarPopup toolsPopup = (CommandBarPopup)toolsControl;
        try
        {
            //Add a command to the Commands collection:
            Command command = commands.AddNamedCommand2(_addInInstance,
                "HelloWorld", "HelloWorld",
                "Executes the command for HelloWorld", true,
                59, ref contextGUIDS,
                (int)vsCommandStatus.vsCommandStatusSupported+
                (int)vsCommandStatus.vsCommandStatusEnabled,
                (int)vsCommandStyle.vsCommandStylePictAndText,
                vsCommandControlType.vsCommandControlTypeButton);
            //Add a control for the command to the tools menu:
            if((command != null) && (toolsPopup != null))
            {
                command.AddControl(toolsPopup.CommandBar, 1);
            }
        }
        catch(System.ArgumentException)
        {
            // If here, the exception is probably because a command with that name
            // already exists. If so there is no need to re-create the command and we
            // can safely ignore the exception.
        }
    }
}

```

The code checks to see the connect mode, and only installs the add-in during the **UI Setup** call. This event is called during the splash screen display when Visual Studio starts.

OnConnection will be called other times by Visual Studio, but there is no need to install the command into the menu structure more than once.

The code will search the IDE's command menu structure to find the **Tools** menu, and then add your add-in to that menu. Most add-in modules are placed on the **Tools** menu, but you are free to put them anywhere you'd like.

Notice that the **Application** parameter is assigned to **_applicationObject**, a private variable in the class. This variable and the **_addInInstance** variable will allow your add-in code to interact with various Visual Studio elements.



Note: The private class variables (**_applicationObject** and **_AddInInstance**) are populated during the connection routine, so they can be referred to during your **Exec()** and **Query Status()** method calls.

Exec Code

The generated code includes an **Exec()** method, which is where you'll add the code you want your add-in to execute. The **Handled** variable, passed by reference, should be set to true to inform Visual Studio that the particular command was processed by the add-in method.

```
public void Exec(string commandName, vsCommandExecOption executeOption, ref object varIn,
ref object varOut, ref bool handled)
{
    handled = false;
    if(executeOption == vsCommandExecOption.vsCommandExecOptionDoDefault)
    {
        if(commandName == "HelloWorld.Connect.HelloWorld")
        {
            MessageBox.Show("Hello World!", "Hello" ,MessageBoxButtons.OK);
            handled = true;
            return;
        }
    }
}
```



Note: You will need to add a reference to **System.Windows.Forms** to include the **MessageBox** code in your add-in.

Query Status code

This method returns a status code to Visual Studio when it requests the current status of your method.

```
public void QueryStatus(string commandName, vsCommandStatusTextWanted neededText,
ref vsCommandStatus status, ref object commandText)
{
    if(neededText == vsCommandStatusTextWanted.vsCommandStatusTextWantedNone)
    {
```

```
        if(commandName == "HelloWorld.Connect.HelloWorld")
        {
            status = (vsCommandStatus)vsCommandStatus.vsCommandStatusSupported
                    |vsCommandStatus.vsCommandStatusEnabled;
            return;
        }
    }
}
```

Although the generated code provides the basic query result, you may need to adjust the code to return a Not Supported status if the add-in module should not be called during the debugging process of Visual Studio.

If your code does not update the status variable, the default behavior is to disable the menu or toolbar item.

Generated files

The wizard will generate the standard project files (Project and AssemblyInfo.cs), as well as the source code files containing your add-in code:

- **Connect.cs:** Generated source code of the add-in.
- **<YourName>.AddIn:** XML Configuration file for your add-in.
- **<YourName> - For Testing.AddIn:** XML configuration file to test your add-in.

Chapter 3 Hooking into the IDE

In this chapter, we will look at the code to hook your add-in module into Visual Studio, and see how you can find the menus and tool windows to integrate with your add-in.

OnConnection Method

The **OnConnection** method is the method used to load your add-in to the Visual Studio IDE. The wizard-generated code searches the GUI controls for the **Tools** menu item and adds your add-in module as the first item in that drop-down menu.



Tip: The wizard will run the connection code when `ConnectMode` is `ext_cm_UISetup`. If you want the add-in module to attach to Tool windows or other items, rather than the standard bar or menu, you might want to wait to connect until `ConnectMode` is `ext_cm_AfterStartup` to ensure the control you want to connect to is created.

Linking to menu items

Visual Studio contains a large collection of commands to perform the IDE functions and a set of controls to provide the user with access to these commands. To link your add-in, you'll need to add your command to Visual Studio's collection and you'll need to add a control into the GUI elements of Visual Studio. We can review how to do these steps by exploring the code in the **OnConnection** method generated by the wizard.

```
Commands2 commands = (Commands2)_applicationObject.Commands;  
string toolsMenuName = "Tools";
```

These two lines put a reference to the command collection into a variable and define the menu (from the top bar) that we want to hook into. You can easily replace the string with the File, Edit, View, Help, or some other menu caption, whichever is the best spot for your add-in. In our example program in [Chapter 5](#), we are going to move our add-in module into the **File** menu, rather than the **Tools** menu.

```
VisualStudio.CommandBars.CommandBar  
    menuBarCommandBar = ((VisualStudio.CommandBars.CommandBars)  
        _applicationObject.CommandBars) ["MenuBar"];  
  
CommandBarController toolsControl = menuBarCommandBar.Controls[toolsMenuName];  
CommandBarPopup toolsPopup = (CommandBarPopup)toolsControl;
```

These next lines get the top menu bar and find the drop-down menu associated with the string we specified previously. The menu control is placed in the **toolsPopup** variable.

At this point, we have both the **commands** collection and the **toolsPopup** GUI control. The following two lines add our add-in to the command collection and the GUI control.

```
Command command = commands.AddNamedCommand2(_addInInstance, "HelloWorld", "Hello World",
    "Classic Hello World example", true, 59, ref contextGUIDS,
    (int)vsCommandStatus.vsCommandStatusSupported+
    (int)vsCommandStatus.vsCommandStatusEnabled,
    (int)vsCommandStyle.vsCommandStylePictAndText,
    vsCommandControlType.vsCommandControlTypeButton);
//Add a control for the command to the tools menu:
if((command != null) && (toolsPopup != null))
{
    command.AddControl(toolsPopup.CommandBar, 1);
}
```

The **AddNamedCommand2()** method has a number of arguments which we can adjust to control our new menu item. After the add-in instance and command name, the next two parameters are the **button text** ("Hello World") and the **tooltip** text ("Classic Hello World example").

The next parameter is the **MSOButton** flag, which indicates how the bitmap parameter is interpreted. The default value of true means the bitmap parameter is an integer ID of a bitmap installed in the application (Visual Studio in this case).

The hard-coded 59 is the **bitmap** parameter which is used to choose the icon to add next to the menu; text.59 is the add-in default icon (a smiley face). However, there are a lot of other options available. A few selected ones are shown in the following code and can be defined as constants in your add-in code.

```
const int CLOCK_ICON = 33;
const int DEFAULT_ICON = 59;
const int EXCEL_ICON = 263;
const int FOXPRO_ICON = 266;
const int TOOLS_ICON = 642;
const int PUSHPIN_ICON = 938;
const int PRINTER_ICON = 986;
const int LIGHTBULB_ICON = 1000;
const int PAPERCLIP_ICON = 1079;
const int DOCUMENTS_ICON = 1197;
const int RED_STAR_ICON = 6743;
```



Tip: There are thousands of icon resources embedded within Visual Studio. You can use a resource editor to preview some of the icons you might want to include on your add-in's menu.

The other parameters are:

- Optional list of GUIDs indicating when the command can be called (typically an empty array is passed).
- Command Status: Typically Supported and Enabled.
- Command Style: How the button is presented (icon and text).
- Control Type: Usually a button control.

You can tweak the command line, for example, to have your add-in module initially disabled and later have your code enable it during the Query Status event.

The **AddControl()** method attaches the newly created command object to the pop-up menu you've chosen. The second parameter, the 1 in this example, refers to the menu position where the new item should be placed.



Note: 1 puts the new object at the top of the menu. You can also get the count of controls on the command bar pop-up menu and add 1, which will put the option at the end of the menu.

Linking to other windows

In addition to the main menu structure, you can also attach your add-in to the various context menus of various IDE windows, such as the **Code** window or the **Solution Explorer**. However, if you do this, you should typically load your command during the AfterStartup connection mode, rather than during UI setup, just to ensure the window you are attempting to attach to is created already in Visual Studio.

```
if (connectMode == ext_ConnectMode.ext_cm_AfterStartup)
{
    ...
}
```

Adding to the code window

The following code sample shows how to add a pop-up menu item to the **Code Window** tool window of Visual Studio. Note we are using **Code Window** rather than **Menu Bar**.

```
// Create the command object.
object[] contextGUIDS = new object[] { };
Commands2 commands = (Commands2)_applicationObject.Commands;

Command cmd = commands.AddNamedCommand2(_addInInstance, "HelloWorld", "Hello World",
    "Hello from Code Window ", true, 59, ref contextGUIDS,
    (int)vsCommandStatus.vsCommandStatusSupported+
    (int)vsCommandStatus.vsCommandStatusEnabled,
    (int)vsCommandStyle.vsCommandStylePictAndText,
    vsCommandControlType.vsCommandControlTypeButton);

// Create a command bar on the code window.
CommandBar CmdBar = ((CommandBars)_applicationObject.CommandBars)["Code Window"];

// Add a command to the Code window's shortcut menu.
CommandBarControl cmdBarCtl = (CommandBarControl)cmd.AddControl(CmdBar,
    CmdBar.Controls.Count + 1);
cmdBarCtl.Caption = "HelloWorld";
```

Note that in this example, we are adding our module to the end of the context menu, not the first item.

We will cover creating an add-in attached to the code window in [Chapter 12](#).

Other IDE windows

There is a large number of other command bar windows you can interact with, including:

- Formatting
- Image Editor
- Debug
- Table Designer

You can find all the available **Command Bar** windows with the following code added to your **Exec()** method.

```
CommandBars commandBars = (CommandBars)_applicationObject.CommandBars;
StringBuilder sb = new StringBuilder();

foreach (CommandBar cb in commandBars)
{
    sb.AppendLine(cb.Name);
}

MessageBox.Show(sb.ToString(), "Windows" ,MessageBoxButtons.OK);
```

The command bar object has both a **Name** and **NameLocal** property (holding localized menu names for international versions of Visual Studio). However, when you search for menus and windows, you can use the English name, which is how they are stored internally.

Adding a toolbar button

The following code sample shows how to add a toolbar button to the standard toolbar of Visual Studio. Note we are using **Standard** instead of **Menu Bar**.

```
// Add the command.
Command cmd = (Command)_applicationObject.Commands.AddNamedCommand(_addInInstance,
    "HelloCommand", "HelloCommand", "Hello World", true, 59, null,
    (int)vsCommandStatus.vsCommandStatusSupported +
    (int)vsCommandStatus.vsCommandStatusEnabled);
CommandBar stdCmdBar = null;
// Reference the Visual Studio standard toolbar.
CommandBars commandBars = (CommandBars)_applicationObject.CommandBars;
foreach (CommandBar cb in commandBars)
{
    if (cb.Name == "Standard")
    {
        stdCmdBar = cb;
        break;
    }
}
// Add a button to the standard toolbar.
CommandBarControl stdCmdBarCtl = (CommandBarControl)cmd.AddControl(stdCmdBar,
    stdCmdBar.Controls.Count + 1);

stdCmdBarCtl.Caption = "Hello World";
// Set the toolbar's button style to an icon button.
CommandBarButton cmdBarBtn = (CommandBarButton)stdCmdBarCtl;
cmdBarBtn.Style = MsoButtonStyle.msoButtonIcon;
```

When adding to a toolbar, the **MsoButton Style** controls how the icon appears on the toolbar. Some options are:

- **msoButtonIcon**: Only show button.
- **msoButtonIconAndCaption**: Show icon and caption text.
- **msoButtonIconAndWrapCaption**: Show icon and wrap caption text.

QueryStatus Method

The **QueryStatus** method is called by Visual Studio whenever the IDE wants to display your menu item. The method returns a status code to the IDE, indicating whether the menu option is currently supported or enabled. Visual Studio then uses this status to determine the menu's appearance and whether or not the user can activate it.

Note that there is no **Command Status Disabled** option. If you want your command to be disabled, simply do not update the status variable, since the default status is disabled.

Other methods

There are other methods you can use to interact with Visual Studio. While these methods are generated as empty modules by the wizard, you might need them depending on your add-in's behavior.

OnAddInsUpdate method

This method is called when add-ins are loaded into the Visual Studio environment (as well as when the user clicks **OK** from the **Add-in Manager** window). If your add-in is dependent on other add-ins, you can check those dependencies during this method.

OnBeginShutdown method

When the user begins to close Visual Studio, this method is called. This is the time to clean up any resources your add-in has created, and save any user configuration information needed for the next time the add-in is loaded.

OnDisconnection method

This method is called when Visual Studio unloads your add-in. If you created or locked any resources when your add-in was connected, this is the method you can use to unlock or free those resources.

On StartupComplete

This method is called once Visual Studio has completed the start-up process. If your add-in is not loaded due to a component dependency, you could install your add-in during this method to ensure all components within Visual Studio have been loaded.

A few caveats

Before we dig in and design some add-in modules, there are a couple of tips to keep in mind.



Tip: *Avoid admin rights. When designing your add-in module, keep in mind that starting with Windows Vista, Windows employs User Account Control (UAC), which means it is very likely*

that Visual Studio will not have admin rights.



Tip: Be careful about storing setting information in non-writable folders or registry entries. Use the APPDATA system variable to find a folder to store your settings.

By keeping the new security model in mind, you can prevent your add-in modules from requiring Visual Studio to be run in admin mode or seeing the access denied error.

Chapter 4 Application and Add-in Objects

In this chapter, we will give a quick overview of the two main object classes that Visual Studio provides to add-in authors to interact with the IDE and with other add-ins.

Application Object

The **_applicationObject** variable contains a DTE2 object reference, which provides properties and methods to allow you to interact with the Visual Studio IDE. Many of these properties will be explored in subsequent chapters and examples. Some of the more commonly used ones are:

ActiveDocument

This property returns a document object reference to the document that currently has focus. The object contains information such as the file name, whether the document has been saved, the selected text, the kind of document being edited, etc.

ActiveWindow

This property returns a window object reference to the currently active window. The window object contains the caption, kind of window (tool or document window), the size (width and height), and position (left and top). It also contains a reference to the document currently in the window. You can do some basic manipulation of the window, such as hiding it, moving it, closing it, etc.

Debugger

This property returns a reference to the debugger object, which allows you to find out the current breakpoints, processes running on the machine, the current program and process, etc. You can also move to various code points, evaluate expressions, etc.

Documents

The **Documents** property is a collection of all currently open documents within Visual Studio. Each individual item refers to a document within the IDE. In [Chapter 11](#), we will work with document objects and their contents.

Edition

This property contains a string indicating the edition of Visual Studio, i.e. Professional, Enterprise, etc. It can be useful if your add-in should not be run in certain editions, for example.

ItemOperations

This property provides an object class that allows you to add new or existing items to the current project. You can also navigate to a URL and have the IDE open a browser window. We will use this object in [Chapter 15](#) when we generate source code files.

LocaleID

This property returns the locale in which the development IDE is running. You might use this to customize your add-in for various countries and languages.

MainWindow

This property returns the main parent window for the IDE. It contains all of the various window properties and you can explore its *LinkedWindows* collection to find the various other windows linked to it.

Mode

The **Mode** property indicates whether the IDE is in design (`vsIDEModeDesign`) or debug mode (`vsIDEModeDebug`). You might want to disable your add-in from running while the user is debugging code.

Solution

This property returns a reference object to the currently open solution in the IDE. The solution object contains a collection of all projects in the solution, the file name, the global settings for the solution, whether it has been saved, etc. In addition, you can add and remove files from the solution, iterate projects, save the solution as another name, etc. We will explore the solution object in [Chapter 8](#).

ToolWindows

This property returns an object that makes it easier to search for some of the common tool windows, such as the Task List, the Solution Explorer, the Error list, etc. We explore tool windows in [Chapter 14](#).

Windows

This property is a collection of windows currently open within Visual Studio. Each item in the collection is a window object, allowing you to resize and move windows, update captions, change focus, etc. We explore the windows collection in detail in [Chapter 10](#).

AddIn Object

The **_addinInstance** object is an instance of the AddIn class. The **_addinInst** parameter is passed to your add-in during the **onConnection** method and it is assigned to the private class variable **_addinInstance**. This variable provides details specific to this instance of your add-in.

Add-in properties

The following properties are available for your add-in.

Collection

The **Collection** property returns a reference to the collection of add-in objects currently installed in Visual Studio. You can use this property to check for any dependencies your add-in may have.

Connected

This is a Boolean value indicating whether your add-in is loaded and connected within Visual Studio. You can connect your add-in programmatically by setting this property to **True** if not already connected, i.e.:

```
if (_addinInstance.Connected)
{
    ...
}
```

Description

This string property contains the descriptive text that is displayed in the Add-in Manager and sometimes as tooltip text. The property is read/write, so you can dynamically update the title in your add-in.

GUID

This read-only string contains the CLSID of the add-in from the add-in registry entry.

Name

This read-only string property holds the command name associated with the add-in. It is the name parameter passed to the **AddNamedCommand** method of the Visual Studio Commands collection.

Object

The **Object** property is a reference to the instance of the actual object containing your add-in's implementation code. You can use this property to access any additional information you've stored in your object class that is needed to implement your add-in module.

ProgID

This read-only string property contains the unique program name for your add-in's command, typically the file name, class name, and command name delimited by periods.

SatelliteDLLPath

This read-only string is the full path name where the DLL containing the code implementing the add-in is located.

Assemblies

The following assemblies are used by the add-in modules and can be added into your code as necessary. Keep in mind that some features are only available in later versions of Visual Studio, so only use them if you know the minimum version your add-in will run in.

Extensibility.dll

This assembly contains all versions of Visual Studio-`IDTExtensibility2` and enums for connection purposes.

CommandBars.dll

Starting in VS 2005, `Microsoft.VisualStudio.CommandBars.dll` contains the command bar model. Early versions used the command bar model from `Office.dll`.

EnvDTE.dll

This assembly contains the extensibility model of Visual Studio to manage the IDE, solutions, projects, files, code, etc. Later versions are all additive to provide more version specific features:

- 80 (VS 2005, 2008, 2010)
- 90 (VS 2008, 2010)
- 100 (VS 2010)

VSLangProj.dll

This assembly contains more detailed extensibility models, specifically for VB.NET and C# projects.

Chapter 5 Save Some Files Add-In

Now that we have explored the various parts of an add-in module, we can put them all together and write a simple add-in project. We can start by creating a basic add-in using the wizard. Be sure to have the wizard generate our starting code and the code to hook it into the Tools menu.

Our add-in is going to look at all documents that have been edited, but not saved, and display them in a check box list. Users can then mark the ones they want to save and click to save only those files. Our add-in will be called `SaveSomeFiles`.

SaveSomeFiles add-in

We can start our add-in using the Add-in Wizard described in [Chapter 2](#). Use the following settings while running the wizard:

- Visual C# (or your preferred language).
- Application Host: Only Visual Studio.
- Name/Description: *SaveSomeFiles* and *Selectively save open files*.
- Create UI Menu and make sure load at start-up is not selected.

Verify the settings in the **Summary** screen, and if they look okay, generate the code.



Note: Add a reference to *System.Windows.Form* in your add-in project's references. You'll need this for the GUI screen we will build. You will want to include this for most add-ins you create.

Designing the selection form

The selection form will be a standard Windows form with a **CheckedListBox** control, **Save**, and **Cancel** buttons. Our add-in will populate the list box and then display it to the user. Once the user clicks **Save**, the code will save the selected files. If the user clicks **Cancel**, the dialog box will close and no action will take place.

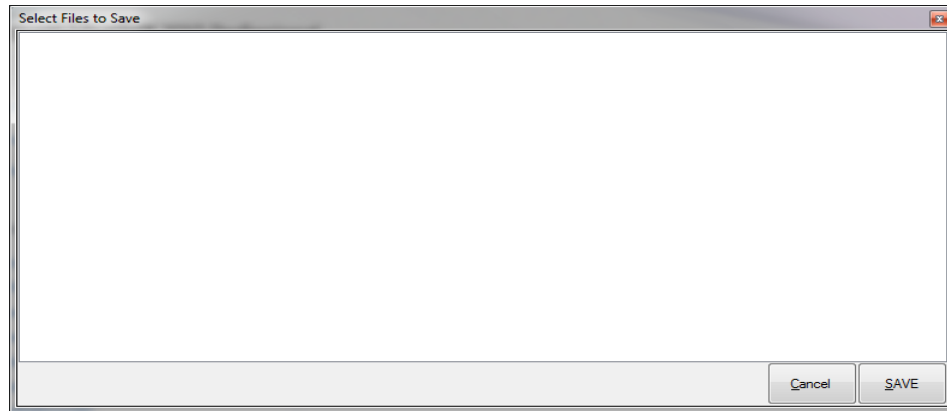


Figure 5: Selection form

Create a Windows form as shown in Figure 5. Name the checked list box control on the form **CLB**. Be sure to set the **Modifiers** property to **Public**, so we can access the checked list box from within our add-in code. In general, any control on the form that will be populated by your add-in will need to be set to public.

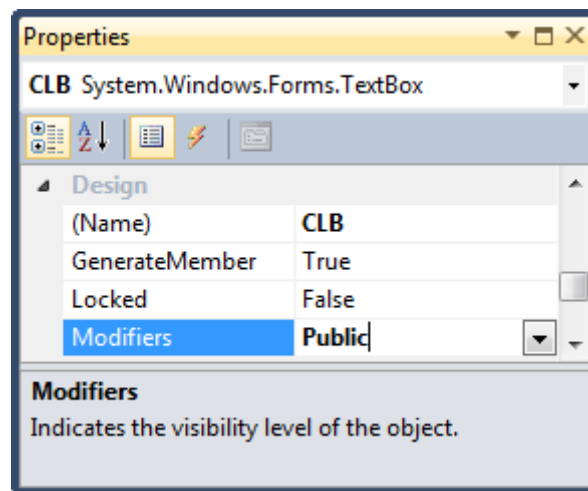


Figure 6: Setting the Modifiers property to Public

Throughout this book, we will create several Windows forms for our add-ins. Feel free to indulge your creative talents to make these screens look nice. The code samples will provide the name and type of control the add-in will interact with. Other than that, we won't spend too much time detailing how to create forms.

Implementing the Exec() method

The code in the **Exec()** method first needs to find out which files need to be saved. It does this by iterating through the documents collection of the **_applicationObject** variable, as shown in the following code sample. Any file that has been modified but has not been saved is added to the check box on the form we created.


```

if(commandName == "SaveSomeFiles.Connect.SaveSomeFiles")
{
    SaveFiles theForm = new SaveFiles();           // Create the form.
    theForm.CLB.Items.Clear();                     // Clear out the items stack.

    // Iterate through each document currently open in the IDE.
    foreach (Document theDoc in _applicationObject.Documents)
    {
        if (theDoc.Saved==false)
        {
            theForm.CLB.Items.Add(theDoc.FullName.ToString());
        }
    }
    // Show the form with "files to be saved".
    theForm.ShowDialog();
}

```

After the user closes the dialog box, we need to step through the checked items, and call the **Save** method on the corresponding document object.

```

if (theForm.DialogResult == DialogResult.OK)
{
    foreach (int xx in theForm.CLB.CheckedIndices)
    {
        foreach (Document theDoc in _applicationObject.Documents)
        {
            if (theDoc.FullName.ToString() == theForm.CLB.Items[xx].ToString())
            {
                theDoc.Save();
            }
        }
    }
}
theForm.Dispose();

```

Once we've completed the **Save** operation for the requested files, we can dispose of the form we created earlier in the code.

But not while debugging

We want to adapt our code so that the **Save Some Files** option is not available if the IDE is in debug mode. To implement this action, we need to update the **Query Status** method.

```

public void QueryStatus(string commandName, vsCommandStatusTextWanted neededText,
                        ref vsCommandStatus status, ref object commandText)
{
    if(neededText == vsCommandStatusTextWanted.vsCommandStatusTextWantedNone)
    {

```

```

        if (commandName == "SaveSomeFiles.Connect.SaveSomeFiles")
        {
            status = (vsCommandStatus)vsCommandStatus.vsCommandStatusSupported |
                    vsCommandStatus.vsCommandStatusEnabled;
        }
    }
    return;
}

```

After we've identified our command (SaveSomeFiles.Connect.SaveSomeFiles), we need to add an additional IF test to determine which status code to return. If we are in debug mode, then we return the default status; otherwise, we return the standard enabled and supported result. The following code can be added into the **Query Status** method.

```

if (commandName == "SaveSomeFiles.Connect.SaveSomeFiles")
{
    if (_applicationObject.Mode == vsIDEMode.vsIDEModeDebug)
    {
        status = (vsCommandStatus)vsCommandStatus.vsCommandStatusSupported;
    }
    else
    {
        status = (vsCommandStatus)vsCommandStatus.vsCommandStatusSupported |
vsCommandStatus.vsCommandStatusEnabled;
    }
    return;
}
else
{
    status = (vsCommandStatus)vsCommandStatus.vsCommandStatusSupported |
vsCommandStatus.vsCommandStatusEnabled;
}
return;
}

```

When the IDE attempts to build the **File** menu, it will ask any add-ins whether they are enabled. If we are in debug mode at the time, the menu option will be disabled.

Summary

In this chapter, we designed a simple add-in module to show how to interact with Windows forms and to extract information from the **_applicationObject** variable. Many add-in modules will have similar approaches, collecting and displaying information to the user, and then interacting through the variable directly with Visual Studio.

Chapter 6 Testing Your Add-In

Once you've coded your add-in, you can easily test it using the Visual Studio debugger. However, debugging can be a bit easier once you review the XML configuration files and understand the add-in life cycle.

Configuration files

When you first create an add-in using the wizard, the wizard generates your class code, and also two XML configuration files that control your add-in's behavior. These are:

- <Add-in Name> - For Testing.Addin
- <Add-in Name>.AddIn

For Testing.AddIn

When you generate an add-in using the wizard, it creates the XML file and places it into the add-in folder. This file allows Visual Studio to load your add-in, but refers to the assembly DLL in your project folder. Other than the assembly location, this file has the same content as your actual AddIn file you'll use to install the add-in.

Of course, this can create a catch-22 for future debugging sessions. When Visual Studio loads the add-in, the DLL containing the add-in code is locked by the IDE. Hence, you might see this message when you attempt to tweak and build your add-in:

Unable to delete file ".\bin\CodeWindow.dll".

Access to the path 'C:\Users\..\documents\visual studio
2010\Projects\CodeWindow\CodeWindow\bin\CodeWindow.dll' is denied.

Add-in settings

The add-in configuration file contains some settings that control how and when the add-in is loaded. These settings can be found in the <Addin> element in the XML file.

LoadBehavior

This value indicates when the add-in is loaded. The available values are:

- 0: Add-in must be loaded manually.
- 1: Add-in automatically loads when IDE starts.
- 4: Add-in loads when started from command prompt.

You will rarely see option 4, since most add-ins provide a UI and would not make sense when run from the command line. Option 0 is good for debugging, because the add-in's DLL is only loaded when you open the add-in, not every time you open the IDE.

CommandPreload

This value determines if the add-in is loaded via the Add-in Manager or automatically when Visual Studio starts (the first time after the add-in file is installed):

- 0: Add-in must be manually started by Add-in Manager.
- 1: Add-in is loaded first time when Visual Studio starts after install.

Sometimes, when you are debugging an add-in, you might need to set **Command Preload** to **0** in the add-in files to allow you to update the DLL when you compile your add-in. If you set it to **1**, you might get the "Unable to delete" error when you attempt to rebuild your add-in file.

I recommend generating the Add-in Wizard without the add-in being loaded at start-up to make it easier for testing and debugging. Once you are ready to deploy your add-in, you can manually change the **Load Behavior** flag to **1** if you want your add-in loaded at start-up time.

Add-in life cycle

The **Load Behavior** setting controls which events from your connect class are called and when. Regardless of the setting, the first two events are always:

- OnConnect with the cm_UISetup connect mode.
- Disconnect with the dm_UISetup disconnect mode.

This is why when you attach your add-in module to Visual Studio during the CM_UISetup mode, it is always called.

0: Call Manually

When set to Call Manually, the add-in does not get called again until you actually request it from the menu. The command has been added to the IDE, and the menu updated, but the code is not loaded. Once you select the item from the menu or toolbar, Connect with cm_AfterStartup is called.

At this point, the add-in is loaded to memory. You can manually unload it using the Add-in Manager, in which case Disconnect with dm_userShutdown is called. If you don't close it manually, the events OnShutdown and Disconnect – with dm_HostShutdown are called. If you've closed it manually, these events will not be called since the add-in is no longer in memory.

If you plan on loading the add-in manually, be sure to set any configuration information you want to save during the Disconnect method.

1: Load at Start-up

When set to load at start-up, additional events are triggered since the IDE is loading your add-in as part of the start-up code:

- Connect with `cm_Startup`.
- Adds-in update event.
- Start-up complete event.

This means that once the IDE makes its appearance, your add-in module is loaded in memory. Unless you unload it manually using the Add-in Manager, the following events will be triggered when the user shuts down Visual Studio:

- Begin Shutdown.
- Disconnect with `dm_HostShutdown`.

The events that your add-in will respond to are handled differently. During debugging sessions, I generally only load the add-in when called from the menu. However, once the code is ready for deployment, I typically set the flag to **1**, load at start-up.

Debugging

When you debug your add-in by pressing F5, a second instance of Visual Studio will be loaded. When this instance loads, it will see the newly added XML file and load the add-in module into Visual Studio. At this point, you can step through the add-in and debug the code or you can run it and test its behavior.

Common mistakes

Here are some common mistakes that might pop up while using your add-in.

Add-in not enabled on menu

If your add-in is not available on the menu, check the **QueryStatus** method and ensure that the return status variable contains both **Command Supported** and **Command Enabled**.

Add-in never invoked

If the command name in the Exec command does not match the add-in class name and friendly name in the add-in XML file, your exec method will never reach your code.

Events not triggering

Be sure the events you are expecting to be called are compatible with the **Load Behavior** mode set in the XML file.

Not seeing code changes

Sometimes, your add-in code appears not to recognize recent code changes. If this is the case, the most likely culprit is that the wrong DLL version was loaded. I would recommend making sure the **Load Behavior** flag is set to **0**, restarting the IDE, and running the add-in from the menu. This should load the most recent version of the DLL.

Removing an add-in module

There are times you might need to remove an add-in module entirely. The easiest way to do this is to find and delete (or rename) the Addin XML file in any of the paths specified in the **Add-in** and **Macros** properties. In the **Tools** menu, select **Options**, and then **Environment**.

Pesky “Unable to delete” message

Occasionally, you might not be able to shake that “Unable to Delete” message, no matter how many times you restart the IDE and tweak settings. If your add-in is not marked as load on start-up, the DLL should not be loaded. However, in the event you cannot unload it, you can start the IDE with the **/SafeMode** switch, which loads Visual Studio without any add-in modules at all.

Even if you start the IDE in **Safe Mode**, you can still debug since the second instance of the IDE will start in regular mode without the **/SafeMode** switch being applied. If you are having trouble working with an add-in, consider making an add-in free shortcut on your desktop to run the IDE with add-ins.

Chapter 7 Visual Studio Environment

In this chapter we will create an add-in module to provide some details about the Visual Studio installed version and the computer that the IDE is currently running on. Although the collected information will be displayed in a Windows form, you could also add logic to create a text file of the information, allowing a user to duplicate the Visual Studio environment on another computer if desired.

VS Info Wizard

Start your VS info add-in by using the wizard and the following settings:

- Visual C# (or your preferred language).
- Application Host: Only Visual Studio, because we wouldn't use this in a macro.
- Name/Description: *VS_Info* and *Info about Visual Studio and Dev Environment*.
- Create UI Menu and load at start-up is not selected.

Verify the settings at the **Summary** screen, and if they look okay, generate the code.

VS Info Form

We need to create a form to hold our Visual Studio information, so we will need to add a Windows form to the project. In addition, we will be using the String Builder object to assemble our information, so add the following line to your **connect.cs** file:

```
using System.Text;
```

Create a form similar to the following figure, but feel free to add your own artistic touches.

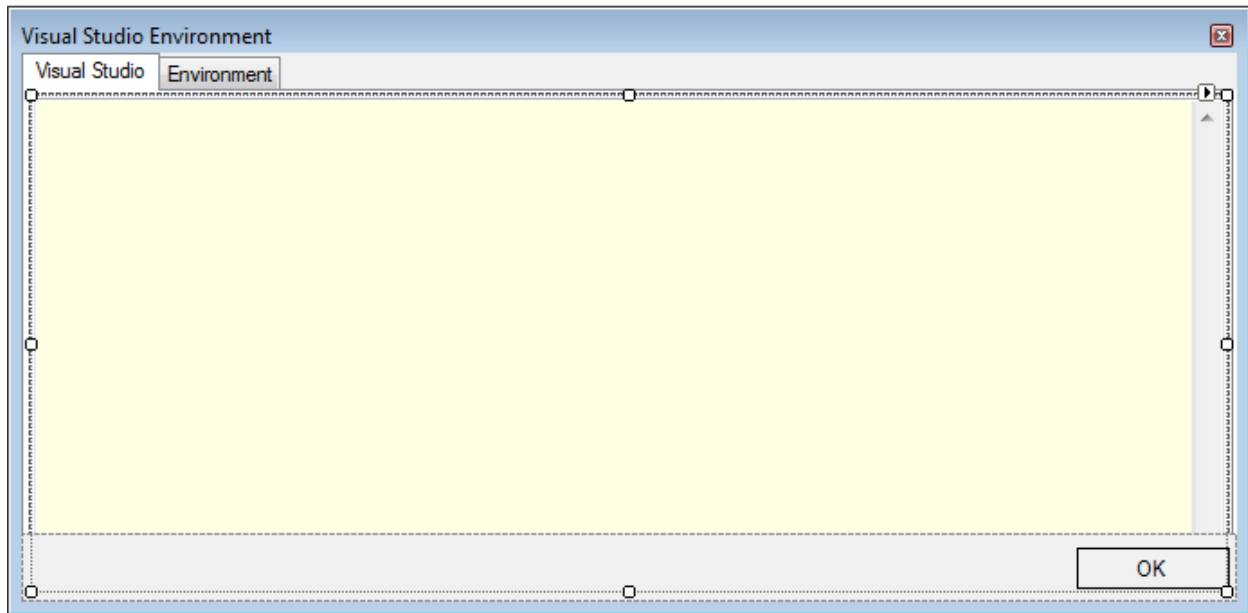


Figure 7: Form for Visual Studio information

However, be sure the two text boxes have PUBLIC modifiers. I've named the Visual Studio text box **VSINFO** and the Environment text box **ENVINFO**. If you use different names, you'll need to tweak the code in your **Exec()** method.

Name the form that you create **VSInfoForm**; I recommend using a monospace font so the generated text will line up nicely.

Exec() method

In our **Exec()** method, we are going to gather information and build lines of text to transfer to the form's windows. We will start simply by grabbing some simple string properties from the **_applicationObject** variable, as seen in the following code:

```
if(commandName == "VS_Info.Connect.VS_Info")
{
    VSInfoForm theForm = new VSInfoForm();           // Create the form.
    StringBuilder sb = new StringBuilder();

    // Get information specifically about Visual Studio.
    sb.AppendLine("Visual Studio " + _applicationObject.Edition + " edition");
    sb.AppendLine("    Version " + _applicationObject.Version.ToString());
    sb.AppendLine("");
    sb.AppendLine("Full EXE Name " + _applicationObject.FullName);
    sb.AppendLine("    Parameters " + _applicationObject.CommandLineArguments.ToString());
    sb.AppendLine("");
    sb.AppendLine("Registry Root " + _applicationObject.RegistryRoot.ToString());
    sb.AppendLine("");
}
```


Getting options

Visual Studio has an options menu to allow you to tweak the settings and behavior of the IDE. It can be found in the **Tools** menu under **Options**.

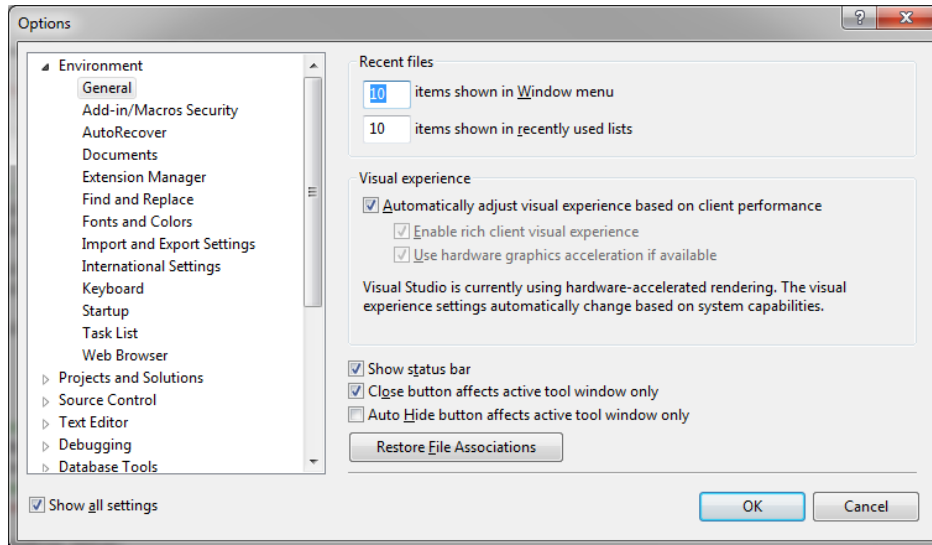


Figure 8: Visual Studio options

You can access any of the Visual Studio options by using the **get_Properties** method of the **_applicationObject** variable. The method takes two parameters, the category name and the page name. The example code that follows shows how to get a collection of the options from the **Environment** category, **General** page.

```
// Gives you access to various IDE options (see Tools | Options menu).  
Properties theSection = _applicationObject.get_Properties("Environment", "General");
```

This method will return a properties collection object with the options from the indicated section. We can then iterate through the properties collection to find the individual options.



Note: You need to know the exact names of the categories and pages; otherwise, you'll encounter an error message.

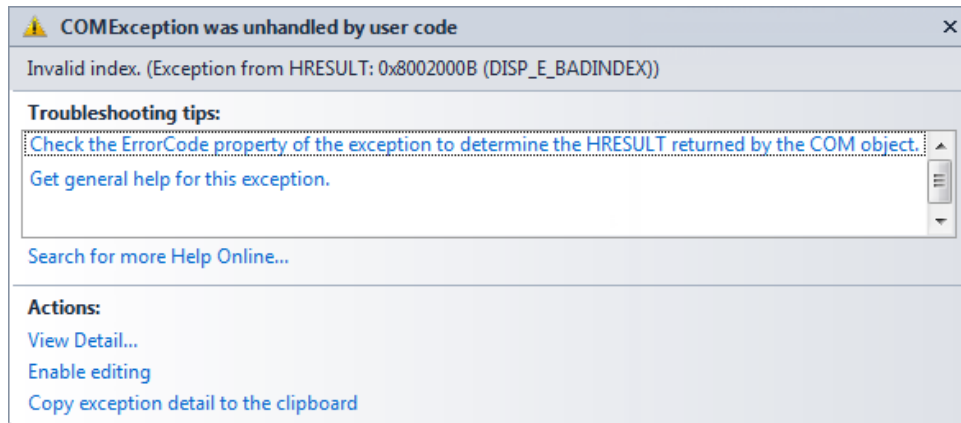


Figure 9: Error result of mismatched option categories and pages

If you plan on using the options, you can go to the following registry key to get the actual category names:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\<ver>\AutomationProperties.

This registry location will show you the actual text fields to use with the **get_Properties** method.

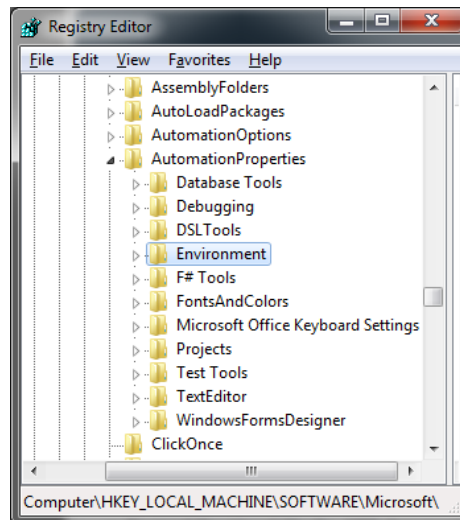


Figure 10: Text fields to use with Get_properties method

For our code, we are going to get the list of folders that Visual Studio looks in when loading add-ins.

```
// Gives you access to various IDE options (see Tools | Options menu).
Properties theSection = _applicationObject.get_Properties("Environment",
                                                         "AddinMacrosSecurity");

foreach (Property theProp in theSection)
{
    // Add-in locations are handled specially.
}
```

```

        if (theProp.Name == "AddinFileLocations")
        {
            object[] theArr = (object[])theProp.Value;
            for (int i = 0; i < theProp.Collection.Count;i++)
            {
                string s = (string)theArr[i];
                sb.AppendLine(s);
            }
        }
    }
}

```

To access the properties, we assign the results of **get_Properties** to a variable of type **Properties**. We then iterate through this collection, getting an individual property object for every entry within the category and page. The property contents will vary a bit, from simple name and value pairs to the slightly more complex code in the previous code sample to iterate the path list for add-ins.

Getting add-ins installed

Once we have gathered the settings and paths, we also want to report on the currently installed add-ins. The following code sample shows how to accomplish that:

```

sb.AppendLine("");
foreach (AddIn theItem in _applicationObject.AddIns)
{
    sb.AppendLine(theItem.Name.ToString()+" (" + theItem.Description.ToString() + ") ");
    sb.AppendLine("    "+theItem.SatelliteDllPath.ToString());
    sb.AppendLine("");
}
// And put the results into the form's edit box.
theForm.VSINFO.Text = sb.ToString();

```

And the final step is to put the string we've just assembled into the form's **Edit** box.

Environment information

We can take a similar approach to provide users some information about the development environment Visual Studio is running in.

```

sb.Clear();
// Get information about development machine.
sb.AppendLine("    Machine Name: "+Environment.MachineName.ToString());
sb.AppendLine("    User: "+Environment.UserDomainName + "/" +
              Environment.UserName.ToString());
sb.AppendLine("Operating System: "+
              OSVersionToFriendlyName(Environment.OSVersion.Version.Major,
              Environment.OSVersion.Version.Minor));

```

```

sb.AppendLine("                "+Environment.OSVersion.ToString() +
                " with " + Environment.ProcessorCount.ToString() +
                " processors");
if (Environment.OSVersion.Platform == PlatformID.Win32Windows ||
    Environment.OSVersion.Platform == PlatformID.Win32Windows)
{
    sb.AppendLine("You are using an older, unsupported OS,
you should consider upgrading to a later version");
}
if (System.Windows.Forms.SystemInformation.MonitorCount > 1)
{
    sb.AppendLine("Multiple monitors setup");
}

theForm.ENVINFO.Text = sb.ToString();

```

Getting an OS-friendly name

You can use the version information supplied in the environment class to convert the version into a friendlier name (such as Windows XP, Windows Vista, etc.) The **OSVersionToFriendlyName()** function handles that task.

```

public string OSVersionToFriendlyName(int MajorVer,int MinorVer)
{
    string OsName = "Unknown";
    switch (MajorVer)
    {
        case 1 : { OsName="Windows 1.0"; break; }
        case 2 : { OsName ="Windows 2.0"; break; }
        case 3:
        {
            switch (MinorVer)
            {
                case 10: { OsName = "Windows NT 3.1"; break; }
                case 11: { OsName = "Windows for Workgroups 3.11"; break; }
                case 5: { OsName = "Windows NT Workstation 3.5"; break; }
                case 51: { OsName = "Windows NT Workstation 3.51"; break; }
            }
        }
        break;
        case 4:
        {
            switch (MinorVer)
            {
                case 0: { OsName = "Windows 95"; break; }
                case 1: { OsName = "Windows 98"; break; }
                case 90: { OsName = "Windows Me"; break; }
            }
        }
        break;
        case 5:

```

```

    {
        switch (MinorVer)
        {
            case 0: { OsName = "Windows 2000 Professional"; break; }
            case 1: { OsName = "Windows XP"; break; }
            case 2: { OsName = "Windows XP Professional x64"; break; }
        }
        break;
    case 6:
    {
        switch (MinorVer)
        {
            case 0: { OsName = "Windows Vista"; break; }
            case 1: { OsName = "Windows 7"; break; }
        }
        break;
    default:
        break;
    }
    return OsName;
}

```

Displaying the form

Once the information has been gathered and transferred to the form, we now simply display the form.

```

theForm.ShowDialog();
handled = true;
return;

```

Final results

Once you build and run the add-in, your screen should look something like this:

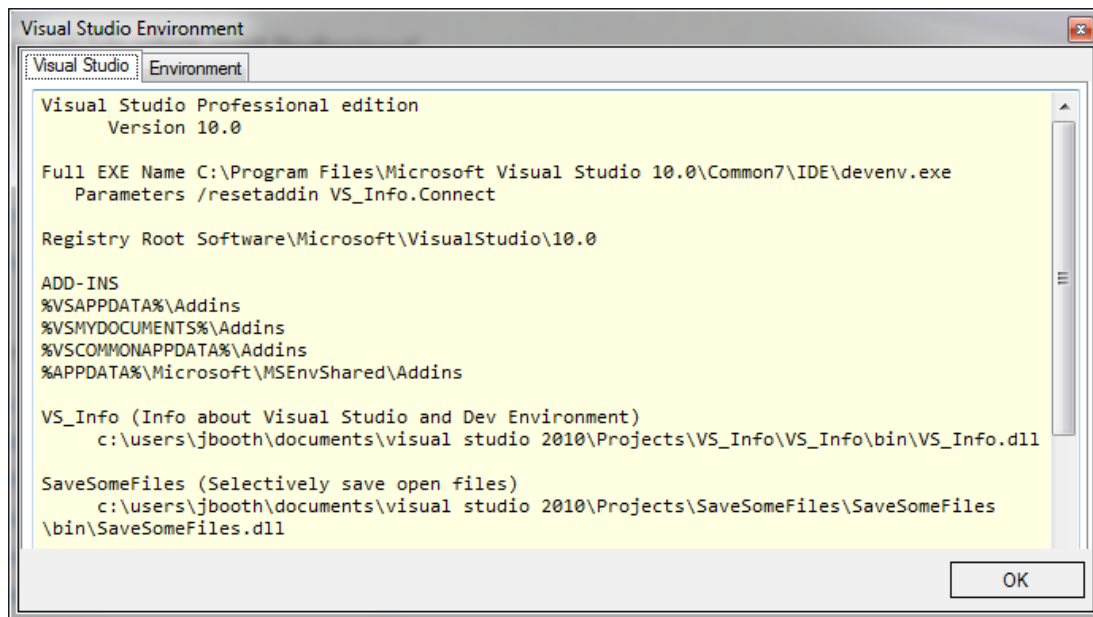


Figure 11: Completed information form

You can adjust the code to include different button options, and even add a print button to print the contents of the Visual Studio setup to a printer or to a text file.

Chapter 8 Solution

In this chapter, we are going to create an add-in module to explore the current solution open in the IDE, and present a summary screen of some key statistics about the solution. It provides a basic example of how to programmatically access various aspects of the solution.

Solution Info Wizard

Start your **Solution Info** add-in using the wizard and the following settings:

- Visual C# (or your preferred language).
- Application Host: Only Visual Studio, since it wouldn't make sense in a macro.
- Name/Description: SolutionInfo, Info, and Statistics *about solution*.
- Create UI Menu and do not load at start-up.

Verify the settings at the **Summary** screen, and if they look okay, generate the code.

Updating the menu icon

To help distinguish our add-in a bit, we are going to change the default icon to the lightbulb symbol. You can add the constant to the top of your **connect.cs** class file:

```
const int LIGHTBULB_ICON = 1000;
```

And revise the **OnConnection** method to use this constant, rather than the hard-code 59 the wizard generates.

```
Command command = commands.AddNamedCommand2(_addInInstance, "SolutionInfo",  
    "Solution Info", "Info & Statistics about solution", true,  
    LIGHTBULB_ICON,  
    ref contextGUIDS,  
    (int)vsCommandStatus.vsCommandStatusSupported +  
    (int)vsCommandStatus.vsCommandStatusEnabled,  
    (int)vsCommandStyle.vsCommandStylePictAndText,  
    VsCommandControlType.vsCommandControlTypeButton);
```

Exec() method

The **Exec()** method will collect a variety of solution information and present it to the user. But first, we need to make sure a solution is open.

```
// Only makes sense if a solution is open.
if (_applicationObject.Solution.IsOpen==false)
{
    MessageBox.Show("No solution is open","ERROR");
    handled = true;
    return;
}
```



Note: Be sure to add a reference to *System.Windows.Forms* to your project.

Solution info

We can now use the **Solution** object to explore some aspects of the currently open solution. We will use a similar approach of building the information in a string builder variable and then transferring the text to our GUI form. In the following code sample, we are taking some of the simple properties of the solution object and displaying them:

```
StringBuilder sb = new StringBuilder();
Solution theSol = _applicationObject.Solution;

sb.AppendLine("          Solution: " + Path.GetFileName(theSol.FullName.ToString()));
sb.AppendLine("          Full Path: " + theSol.FullName.ToString());
sb.AppendLine("Start-up projects: ");
foreach (String s in (Array)theSol.SolutionBuild.StartupProjects)
{
    sb.AppendLine("          " + s);
}
```

Totaling project information

We also want to report on the number of projects and code files the solution contains. To do so, we need to iterate through the projects associated with the solution.

```
Project theProj;                                // Generic project item.
int NumVBprojects = 0;
int NumVBmodules = 0;
int NumCSprojects = 0;
```



```

int NumCSmodules = 0;
int NumOtherProjects = 0;

// Iterate through the projects to determine number of each kind.
for (int x = 1; x <= theSol.Count; x++)
{
    theProj = theSol.Item(x);
    switch (theProj.Kind)
    {
        case PrjKind.prjKindVBProject :
        {
            NumVBprojects++;    // Increment number of VB projects.
            NumVBmodules += theProj.ProjectItems.Count;
            break;
        }
        case PrjKind.prjKindCSharpProject :
        {
            NumCSprojects++;    // Increment number of C# projects.
            NumCSmodules += theProj.ProjectItems.Count;
            break;
        }
        default:
        {
            NumOtherProjects++;
            break;
        }
    }
}

```

The **Kind** property of the project object is a GUID string, so we need a little help in determining the project type. To this end, we need to add a reference to **VSLangProj** into our add-in project.

You might see a compiler error stating that prjKind cannot be embedded when you attempt to reference the project kinds. To solve this error, right-click on the **VSLangProj** reference and bring up the **Properties** dialog. Set the **Embed Interop Types** to **false** to prevent the types from being embedded in the assembly.

Now that we've collected the information, we need to format it for display to our user, which the following code sample shows:

```

sb.AppendLine("Visual Basic code");
sb.AppendLine("      " + NumVBprojects.ToString() + " projects containing " +
              NumVBmodules.ToString() + " modules");
sb.AppendLine("");
sb.AppendLine("Visual C# code");
sb.AppendLine("      " + NumCSprojects.ToString() + " projects containing " +
              NumCSmodules.ToString() + " modules");
sb.AppendLine("");
sb.AppendLine("Miscellaneous projects");
sb.AppendLine("      " + NumOtherProjects.ToString() + " other projects");
sb.AppendLine("");

```

Project GUIDS are stored in a registry key, HKLM\Software\Microsoft\VisualStudio\<vers> Projects. You can copy the GUID to additional constants if you need to report on other project types during the **Solution** add-in. The following are some sample constants for VS projects:

```
// Constants for additional project types.
const string WEB_APPLICATION_PROJECT = "{E24C65DC-7377-472b-9ABA-BC803B73C61A}";
const string DEPLOYMENT_PROJECT = "{54435603-DBB4-11D2-8724-00A0C9A8B90C}";
```

Properties

After we've gathered some of the solution info, we can iterate through the properties associated with the solution, and append them to our string builder variable.

```
// Get properties.
sb.AppendLine("Solution Properties");
Properties props = theSol.Properties;
foreach (Property prop in props)
{
    sb.Append("    " + prop.Name + " = ");
    try
    {
        sb.AppendLine(prop.Value.ToString());
    }
    catch
    {
        sb.AppendLine("(Nothing)");
    }
}

// Put built string onto form.
SolInfoForm theForm = new SolInfoForm();
theForm.SOLINFO.Text = sb.ToString();
theForm.ShowDialog();

handled = true;
return;
```

Displaying the results

After we've built our string builder variables, we need to create a form to display them to the end user.

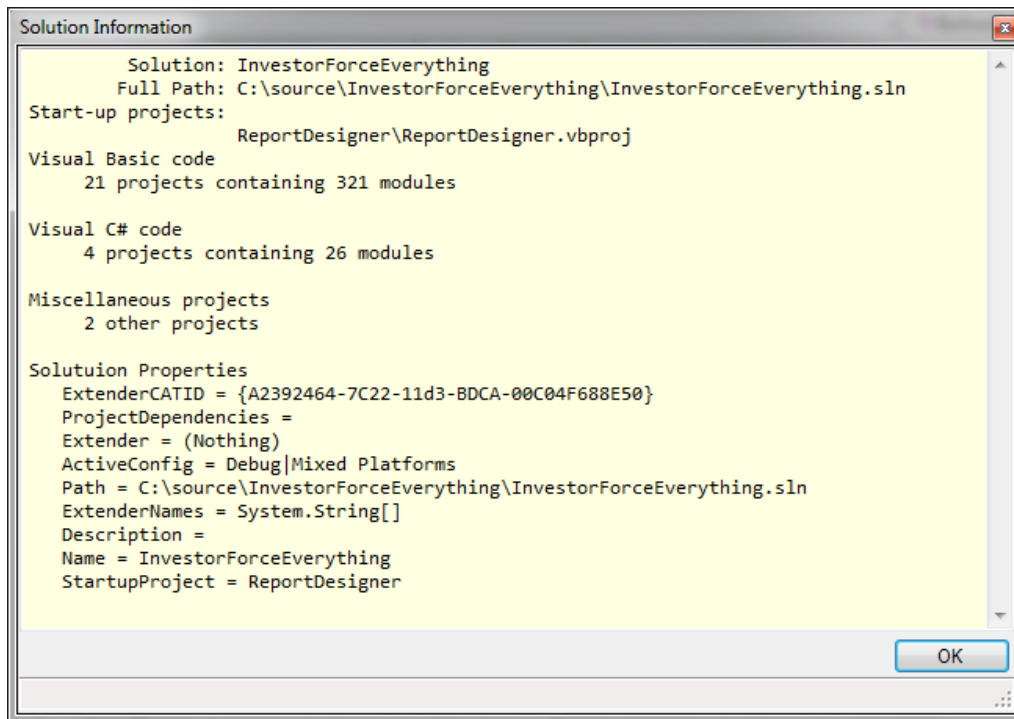


Figure 12: Solution Information form

You can download the project code or create your own form similar to the one in Figure 12. The add-in code assumes the text box control is named SOLINFO. Be sure to name it the same and mark its modifier as PUBLIC so the add-in can place the solution information on the form.

Solution methods

In addition to displaying information about the solution, you can also perform certain operations on the solution, much like the IDE does. Some of these include:

Close

This method closes the solution, with an optional Boolean parameter to save the solution first.

FindProjectItem

This method searches the project space, looking for an item by file name. If the item is found, the method returns a ProjectItem reference to the file you were searching for.

SaveAs

This method allows you to save the solution under a different file name.

SolutionBuild

The solution object also provides information about the active configuration and the last build state. You can use the **Solution Build** object of the solution to perform solution-level operations, such as:

Build

Build the solution with an optional parameter to wait for the build to complete. You might decide to do automated solution builds in the background as part of your testing cycle.

Clean

This method cleans up extra files used by the solution, and features an option to wait for completion before continuing.

Run

This option runs the start-up project associated with the solution.

BuildState

This property reports the current state of the build and is an enumerated type from the following list:

- `vsBuildStateDone`: Build is complete.
- `vsBuildStateInProgress`: Solution currently being built.
- `vsBuldStateNotStarted`: Solution has not been built.

Chapter 9 Projects

In this chapter, we are going to create an add-in module which will generate an HTML document providing technical details about the project. It will save the HTML file to disk and open it in a browser window to be viewed from within Visual Studio.

Project Info Wizard

Start your **Project Info** add-in using the wizard and the following settings:

- Visual C# (or your preferred language).
- Application Host: Only Visual Studio.
- Name/Description: *ProjectInfo* and *Generate HTML project documentation*.
- Create UI Menu and do not load at start-up.

Verify the settings at the **Summary** screen, and if they look okay, generate the code.

Exec() method

The **Exec()** method will collect the projects associated with the solution and present them to the user, but first, we need to make sure a solution is open.

```
// Only makes sense if a solution is open.
if (!_applicationObject.Solution.IsOpen==false)
{
    MessageBox.Show("No solution is open", "ERROR");
    handled = true;
    return;
}
```



Note: Be sure to add a reference to *System.Windows.Forms* to your project. You will also need to add *VSLangProj* and disable the *Embed Interop types* property.

Once we know we have an open solution, we can start to build HTML documentation.

```
// Find all project information.
Solution theSol = _applicationObject.Solution;
StringBuilder sb = new StringBuilder();
string shortName = Path.GetFileNameWithoutExtension(theSol.FullName);

sb.AppendLine("<html>");
sb.AppendLine("<head>");
```

```

sb.AppendLine("<title>" + shortName + "</title>");
AddJavaScript(sb);
sb.AppendLine("</head>");
sb.AppendLine("<body>");
sb.AppendLine("<h1>" + shortName + " solution</h1>");

```

Notice the **AddJavaScript()** function in the middle, which we will use to write some simple JavaScript functionality in our webpage. We will add this function toward the end of the chapter.



Tip: If you want to learn JavaScript quickly, be sure to download [JavaScript Succinctly](#) from the Syncfusion website. [jQuery Succinctly](#) is another excellent reference book.

Getting each project

Using the solution object as a starting point, we can write code that will loop through all projects and output some basic project information to the HTML file we are building.

```

VSProject theVSProj = null;
Project theProj;
for (int xx=1;xx<=theSol.Projects.Count;xx++)
{
    theProj = theSol.Projects.Item(xx);
    theVSProj = (VSProject)theSol.Projects.Item(xx).Object;
    sb.AppendLine("<h2 onclick='ToggleDiv(\"proj\"+xx.ToString()+\" \");'>" +
        theProj.Name+"</h2>");
    sb.AppendLine("<div id='proj' + xx.ToString() + \" style='display:none;'>");
    sb.AppendLine("<h3 onclick='ToggleDiv(\"info\" + xx.ToString() + \" \");'>INFO</h3>");
    sb.AppendLine("<div id='info' + xx.ToString() + \">");
    sb.AppendLine("<p>Unique Name: " + theProj.UniqueName + "</br>");
    sb.AppendLine("Full Path: " + theProj.FullName + "</br>");
    // Report language
    if (theProj.Kind==VSLangProj.PrjKind.prjKindCSharpProject)
    { sb.AppendLine(" Language: C#</p>"); }
    if (theProj.Kind == VSLangProj.PrjKind.prjKindVBProject)
    { sb.AppendLine(" Language: Visual Basic</p>"); }
    sb.AppendLine("</div>");
}

```

We get the **Item()** project and create two project variables from it. The first, the **Project** object type, is a generic project reference, providing us with basic information such as name, path name, unique name, etc. The second variable, the **VSProject** object type, is a Visual Studio project, and contains additional properties and methods unique to Visual Studio.

Project type

The project type has general properties we can use to display the project info:

- Name: Short name of project.
- FullName: Full name and path of project.
- Unique Name: Namespace and unique project name.

VSPProject type

The VSPProject type has more detailed information, specifically in Visual Studio. We can access this information to display all the references that the project uses. The following code sample places the references into a list structure in our HTML documentation.

References

This example code includes the reference name and description, version number, and whether the reference is an ActiveX control or an assembly.

```
sb.AppendLine("<h3 onclick='ToggleDiv(\"ref\" + xx.ToString() + \"\");'>REFERENCES</h3>");
sb.AppendLine("<div id='ref\" + xx.ToString() + \"'>");
sb.AppendLine("<ul>");

foreach (Reference theRef in theVSProj.References)
{
    string theVer = theRef.BuildNumber.ToString();
    if (theVer == "0")
    {theVer = theRef.MajorVersion.ToString() + "." + theRef.MinorVersion.ToString(); }

    sb.Append("<li>" + theRef.Name + " (" + theVer + ")");
    if (theRef.Description.Length > 0)
    { sb.Append(" -" + theRef.Description); }

    if (theRef.Type == prjReferenceType.prjReferenceTypeActiveX)
    { sb.Append(" [ActiveX] "); }
    sb.AppendLine("</li>");
}
sb.AppendLine("</ul>");
sb.AppendLine("</div>");
```

Project Items

Each project contains a list of all of the items that make up that project. These can be accessed via the **Project Items** property. The project items consist of the various source code files that make up the project. These can include source files, XML files, etc. If the file is a source code file, Visual Studio provides a code model which allows our code to access the namespace, classes, etc. within that file. We will use the code model to show the classes with a source file in this chapter, but cover the code model in more depth in [Chapter 13](#).

```

sb.AppendLine("<h3 onclick='ToggleDiv(\"items\" + xx.ToString() + \"\");'>ITEMS</h3>");
sb.AppendLine("<div id='items\" + xx.ToString() + \"' style='display:none;'>");
sb.AppendLine("<ul>");
FileCodeModel theCM = null;

foreach (ProjectItem theItem in theProj.ProjectItems)
{
    sb.AppendLine("<li>" + theItem.Name);
    theCM = theItem.FileCodeModel;
    if (theCM != null)
    {
        sb.AppendLine("<ul>");
        foreach (CodeElement theElt in theCM.CodeElements)
        {
            // List all the classes we find within the code file.
            if (theElt.Kind == vsCMElement.vsCMElementClass)
            {
                sb.AppendLine("<li>" + theElt.Name + "</li>");
            }
            // If we find a namespace, there may be a class in there as well.
            if (theElt.Kind == vsCMElement.vsCMElementNamespace)
            {
                foreach (CodeElement theInnerElt in theElt.Children)
                {
                    string theNameSpace = theElt.Name;
                    if (theInnerElt.Kind == vsCMElement.vsCMElementClass)
                    {
                        sb.AppendLine("<li>" + theNameSpace + "/" +
                            theInnerElt.Name + "</li>");
                    }
                }
            }
        }
        sb.AppendLine("</ul>");
    }
    sb.AppendLine("</li>");
}
sb.AppendLine("</ul>");
sb.AppendLine("</div>");

```

For every project item, we first include the name in the list we are building. We also check to see if a code model exists for the current file (which one should for C# and VB modules). If we find a code model, we iterate through the code elements, looking for either class entities or classes within namespaces. For each class we encounter, we include the class name and optionally the namespace.

This allows our project HTML display to drill down to the class level for a given project. A sample list of items that will be generated is shown in the following list. This is a C# project that has assembly information and two source files. One file has a single class called **ComputePayrollAmount** and the other source file has a class called **Connect** within a namespace called **CodeModelSample**:

ITEMS

- AssemblyInfo.cs

- Class1.cs
 - ComputePayrollAmount
- Connect.cs
 - CodeModelSample/Connect

If your add-in is going to do any type of source code manipulation, be sure to explore the code model object described in [Chapter 13](#).

Adding the JavaScript

The following function adds the JavaScript to the HTML header to allow us to toggle various project elements.

```
private void AddJavaScript(StringBuilder sb)
{
    sb.AppendLine("<script type='text/javascript'>");
    sb.AppendLine("function ToggleDiv(theID) ");
    sb.AppendLine("{");
    sb.AppendLine("var e = document.getElementById(theID);");
    sb.AppendLine("if(e.style.display == 'block')");
    sb.AppendLine("    e.style.display = 'none';");
    sb.AppendLine("else");
    sb.AppendLine("    e.style.display = 'block';");
    sb.AppendLine("}");
    sb.AppendLine("</script>");
}
```

Showing the Results

Once the HTML file is generated, the following code saves it and then displays it.

```
string theFile = Path.ChangeExtension(theSol.FullName, "html");
System.IO.File.WriteAllText(theFile, sb.ToString());
System.Diagnostics.Process.Start("file://" + theFile);
```

Styling the HTML

The generated HTML is rather plain looking, but you can add some style sheet commands to improve the look and feel of the document. The following code is a function to add style commands to the HTML document.

```
private void AddStyles(StringBuilder sb)
{
    sb.AppendLine("h2 {");
    sb.AppendLine("font: bold italic 2em/1em \"Times New Roman\",");
    sb.AppendLine("    \"MS Serif\", \"New York\", serif;");
    sb.AppendLine("margin: 0;");
    sb.AppendLine("padding: 0;");
    sb.AppendLine("border-top: solid #e7ce00 medium;");
    sb.AppendLine("border-bottom: dotted #e7ce00 thin;");
    sb.AppendLine("width: 600px;");
    sb.AppendLine("color: #e7ce00;");
    sb.AppendLine("}");
}
```

In you want to use such a function, insert the function call immediately after the **AddJavaScript()** function call.

Chapter 10 IDE Windows

In the prior chapters, we've looked at the Visual Studio environment and the solution and projects that a user can edit. In this chapter, we are going to explore the open windows and files that a user can be working with in Visual Studio at any given time.

Windows

When you open Visual Studio, even without a solution open, a number of windows are created by the application for helping with the development tasks. These are referred to as your tool windows. When you edit a source code file or a form, these are open in an editor window, and referred to as document windows. You can do some interaction with these windows, such as resizing them, moving them, activating and closing them, etc.

Tool windows

When you open Visual Studio, the following tool windows are generally open even before you open a solution.

- Solution Explorer
- Start Page
- Properties
- Find and Replace
- Object Browser
- Class View

You can find the associated window by stepping through the **Windows Collection** property on the application object. The window object will have basic window manipulation properties, but you can cast it to a specific window type, such as the Solution Explorer, the Error List, etc. We will cover some of the tool windows in more detail in [Chapter 14](#).

Document windows

Once you open a solution in Visual Studio, every open source-file will be placed into a document window. The type of window varies depending on the type of file being edited. The **Code Window** holds source code files, such as VB or C# code. HTML or ASPX files have another editor, visual design tools, etc.

Window object

The **Window** object contains properties and methods to do basic manipulation of the window itself, not its contents.

Properties

You can manipulate the window's location and behavior using the coordinates and some Boolean options:

- `AutoHides(Boolean)`: Sets whether or not the window can be hidden.
- `Caption(string)`: Caption on window title bar.
- `Document(object)`: Associated document object if “document” window.
- `Height and Width (integer)`: Window size in pixels.
- `Left and Top (integer)`: Window location from edge of container.
- `Visible (Boolean)`: Sets whether or not the window is currently displayed to the user.

Methods

You can manipulate the window using a few methods:

- `Activate`: Give the window focus.
- `Close`: Close the document with the option to save it.

ActiveWindow

The **ActiveWindow** property returns a reference to the window object that currently has focus in Visual Studio. Typically, this will be a document window containing the code currently being edited, but can be any window the user clicked on. If no solution is open, the **ActiveWindow** will return the main window. You can test the **Kind** property to see if a “document” or “tool” window is currently active.

MainWindow

The main window is a special window distinct from other windows in the environment. It is typically maximized and the docking window for all the other tool and document windows the user might have open. The Boolean properties of **AutoHides** and **IsFloating** are not defined for the main window.

The main window's **Kind** property will be **Tool** and its **Type** property will be **vsWindowTypeMainWindow**.

Windows

The windows collection contains a list of all windows in the IDE (some of which may not be visible). You can iterate through this list or search for a particular window using the **vsWindowsKind** constants. For example, the following code sample gets a few selected windows and saves them to variables for easier reference:

```
DTE2 theApp = _applicationObject;
Window theSolExploer = theApp.Windows.Item(Constants.vsWindowKindSolutionExplorer);
Window theProperties = theApp.Windows.Item(Constants.vsWindowKindProperties);
Window theCallStack = theApp.Windows.Item(Constants.vsWindowKindCallStack);
```

You can also walk through the windows collection, or search for all document windows as the following code sample illustrates:

```
foreach (Window theWind in _applicationObject.Windows)
{
    if (theWind.Kind == "Document")
    {
        MessageBox.Show(theWind.Caption);
    }
}
```

This can be useful if you want your add-in to provide an option to scan all open documents rather than the entire project.

Window Kind constants

The following **WindowKind** constants are available to find particular windows in the collection:

- **vsWindowKindCallStack**: The Call Stack window.
- **vsWindowKindClassView**: The Class View window.
- **vsWindowKindCommandWindow**: The Command window.
- **vsWindowKindFindReplace**: The Find Replace dialog.
- **vsWindowKindFindResults1**: The Find Results 1 window.
- **vsWindowKindMainWindow**: The Visual Studio IDE window.
- **vsWindowKindObjectBrowser**: The Object Browser window.
- **vsWindowKindOutput**: The Output window.

- `vsWindowKindProperties`: The Properties window.
- `vsWindowKindResourceView`: The Resource Editor.
- `vsWindowKindServerExplorer`: The Server Explorer.
- `vsWindowKindSolutionExplorer`: The Solution Explorer.
- `vsWindowKindTaskList`: The Task List window.
- `vsWindowKindToolbox`: The Toolbox.
- `vsWindowKindWatch`: The Watch window.

You can use these constants to find a particular window, or test the **ObjectKind** property of a window against the constant to know the window you are interacting with, as follows:

```
if (theWind.ObjectKind == EnvDTE.Constants.vsWindowKindCallStack)
{ // Do something with call stack. }
```

Tool windows

Several of the various tool windows can be cast to object types to allow properties and methods specific to that particular tool. These include:

```
CommandWindow theCMD      = _applicationObject.ToolWindows.CommandWindow;
ErrorList theErrs         = _applicationObject.ToolWindows.ErrorList;
UIHierarchy theSolExplore = _applicationObject.ToolWindows.SolutionExplorer;
OutputWindow theOutput    = _applicationObject.ToolWindows.OutputWindow;
TaskList theTaskList      = _applicationObject.ToolWindows.TaskList;
ToolBox theToolBox        = _applicationObject.ToolWindows.ToolBox;
```

The tool window types are explored in more detail in [Chapter 14](#).

Document windows

In addition to the supporting tool windows, there are a number of different editors that might be used when a source file is open. The most common is the code window (which we discuss in [Chapter 12](#) and [Chapter 13](#)); however, you can use the window object to gather some information about the source code being edited.

When you obtain a window's object, either through the **ActiveWindow** property or by searching the Windows collection, you can use the **Kind** property of "document" to know it is a source file being editing in the IDE. The window will also have an **Object** property associated with it, and you can test the type of this property to determine what kind of source window is being looked at. For example, the following code can test whether Visual Studio is looking at some sort of HTML code:

```
Window ActiveWin = _applicationObject.ActiveWindow;    // Grab the active window.
if (ActiveWin.Object is HTMLWindow)
{
    HTMLWindow theHTML = (HTMLWindow)ActiveWin.Object; // Cast as an HTML window.
}
```

You can use the Visual Basic Assemblies, which are automatically included in VB projects but need to be added manually to C# projects, to determine the type of a window. The following code shows the type of window, which you can then cast the object to:

```
Microsoft.VisualBasic.Information.TypeName(ActiveWin.Object)
```

If you want to have your add-in manipulate different kinds of windows, this can help you find the window type to cast the object property to.

Is AJAX being used?

For a simple example of how to use the window object, we are going to write an add-in that will determine whether the HTML or ASPX code currently open in the active window appears to be using AJAX. AJAX requires a script manager object and at least one Update Panel. This wizard will look at the HTML code in the designer window and see if both elements are found.

We are still going to use the wizard to create our basic add-in, so fire up the wizard with the following:

- Visual C# (or your preferred language).
- Application Host: Only Visual Studio.
- Name/Description: IsAjaxEnabled and *Check for HTML and AJAX*.
- Create UI Menu and do not load at start-up.

Although this is a simple wizard, it will show the basics of how to parse HTML code being edited in Visual Studio.

Getting the active window

In our **Exec()** method of the code, we need to get the active window and make sure it is not a tool window. The following code does that:

```
if(commandName == "IsAjaxEnabled.Connect.IsAjaxEnabled")
{
    handled = true;
    Window ActiveWin = _applicationObject.ActiveWindow;    // Grab the active window.

    if (ActiveWin.Kind != "Document")
    {
        // Tell user only wants on document windows.
        MessageBox.Show("Please select an HTML or ASPX page to check....");
        return;
    }
}
```



Note: Don't forget to add the reference to *System.Windows.Forms* for the *MesageBox()*.

Making sure it is HTML code

Once we know it is a document window, we want to confirm it is a HTML window and if so, typecast the window object to HTML Window.

```
// And only for HTML modules (ASPX, HTML, etc.).
if (ActiveWin.Object is HTMLWindow)
{
    HTMLWindow theHTML = (HTMLWindow)ActiveWin.Object;    // Cast as an HTML window.
    Boolean FoundSM = false;
    Boolean FoundUP = false;
}
```

We are also setting up our flags to check the AJAX code samples we are searching for.

Parsing the HTML code

We can write our own HTML code parser if we are feeling particularly ambitious, but to keep things simple, we will use Microsoft's parser instead.

The HTML Window object we created previously has a property called **CurrentTab**, which can be:

- vsHTMLTabsDesign
- vsHTMLTabsSource

If the HTML window is on the design page, the **CurrentTabObject** property will contain a reference to the HTML document object from the page. So we are going to save the user's current mode, and if need be, switch to the design tab so we can grab that HTML document for our parsing purposes.

```
vsHTMLTabs PriorMode = theHTML.CurrentTab;
// See if in Design mode, and if not, switch to it.
if (theHTML.CurrentTab != vsHTMLTabs.vsHTMLTabsDesign)
{
    theHTML.CurrentTab = vsHTMLTabs.vsHTMLTabsDesign;
}
if (theHTML.CurrentTab == vsHTMLTabs.vsHTMLTabsDesign)
{
    // Get an HTML document from the current object in the design window.
    IHTMLDocument2 theHTMLDoc = (IHTMLDocument2)theHTML.CurrentTabObject;
```



Note: You'll need to add a reference to *Microsoft.mshtml* to create the HTML document object.

Once you have the HTML document object, you can use the object to walk through the entire document model. For our code, we are simply searching each element to see if we find a script manager and an update panel.

```
foreach (IHTMLElement element in theHTMLDoc.all)
{
    try
    {
        if (element.outerHTML.ToUpper().Contains("ASP:SCRIPTMANAGER"))
        { FoundSM = true; }
        if (element.outerHTML.ToUpper().Contains("ASP:UPDATEPANEL"))
        { FoundUP = true; }
    }
    catch
    {
    }
}
```

There is a lot of functionality built into the HTML document object. You could simply set the **BGColor** property and the IDE will add the appropriate element to the document and mark the source HTML file as edited. The scope of the HTML document is beyond this book, but can be a very useful starting point if you want to manipulate HTML or ASPX code opened in Visual Studio.

Showing our findings

Once we've made our loop through the HTML elements, we want to return the designer back to its original mode, and then report our findings.

```

if (theHTML.CurrentTab != PriorMode)
{
    theHTML.CurrentTab = PriorMode;
}
// Evaluate flags to see if we are using AJAX.
if (FoundSM && FoundUP)
{
    MessageBox.Show("AJAX appears to be in use on this form");
}
else
{
    if (FoundSM)
    {
        MessageBox.Show("Script Manager found, but no update panels...");
    }
    if (FoundUP)
    {
        MessageBox.Show("Update Panel(s) found, but missing Script Manager...");
    }
}
if (FoundSM == false && FoundUP == false)
{
    MessageBox.Show("AJAX does not appears to be used on this form");
}

```

Summary

While this add-in module performs a very simple task, it does provide an example of how to easily parse HTML code. Microsoft .NET provides a nice collection of tools to manipulate HTML, and plugging that into the windows of the add-in code should give you a good starting point to develop tools for your HTML code.

Chapter 11 Documents

Each source file being edited is represented in Visual Studio as a document object, which has the necessary properties to access the file content, save it, etc. In this chapter, we will look at how to use the document object.

Getting the document

The document object can be obtained from any source window or by asking Visual Studio for the active document. Any “Document” type window will include a reference to the document object associated with it. The following code sample illustrates a few ways to get the document object.

```
Document ActiveDoc = _applicationObject.ActiveDocument;

foreach (Document CurrentDoc in _applicationObject.Documents)
{
}

foreach(Window CurWindow in _applicationObject.Windows)
{
    if (CurWindow.Kind == "Document")
    {
        Document CurDoc = CurWindow.Document;
    }
}
```

Document object

The document object has some basic properties to allow you to determine the file and path name, which windows the document is loaded in, which project item it is associated with, etc.:

- **ActiveWindow:** Window the document object is actively displayed in.
- **FullName:** Path and file name of document.
- **Language:** String containing language, e.g., CSHARP, CSS, XML, VB, etc.
- **Path:** Folder document is located in.
- **ProjectItem:** Associated project item the document is from.
- **Saved:** Has the document been changed since last opened?
- **Selection:** Selected text object associated with the document.
- **Windows:** All windows the document is displayed in.

With these basic properties, you can navigate from the document to windows or to the project item within the solution. You could back up a copy of the document before you make any changes, etc.

In addition, there are a few methods you can use with the document object to save the file, activate its window, close the document, etc.:

- `Activate()`: Move focus to this document.
- `Close()`: Close with the option to save.
- `Redo()`: Redo the last operation.
- `Save()`: Save the document with an optional "Save As" file name.
- `Undo()`: Undo the last operation.

You can use the basic properties and methods to perform operations on the document as a whole. In later chapters, we will explore getting the code and text from the document and manipulating it as well.

Text document object

Each document object has an object method which provides access to the text content of the document and allows you to do some basic edit operations in the document. You can get the associated text document with the following code sample.

```
Document theDoc = _applicationObject.ActiveDocument;  
TextDocument theTextDoc = (TextDocument)theDoc.Object("TextDocument");
```

The text document object has two properties of interest to help editing the text in the document window. The start and end points are objects representing the first and last points in the file. You can create an edit point from either of these objects if you want to do some basic editing. We cover how to edit with edit points in the next chapter. An edit point is the programmatic equivalent of the user's cursor location while editing. Edit operations take place from the edit point in the document.

The text document has some basic manipulation methods for the document. These include:

- `ClearBookMarks()`: Clear any bookmarks from the document's margin.
- `CreateEditPoint()`: Position the "programmatic" cursor for editing.
- `MarkText()`: Search for a pattern and mark lines containing the pattern.
- `ReplacePattern()`: Search and replace text in the document.

These methods allow you to easily manipulate the text content.

Converting C# to VB

There is a great number of websites that will convert your C# code to VB.NET, but let's assume we wanted to tackle such a beast ourselves (which is way beyond the scope of this book). The following code sample shows how we could use the **MarkText** and **ReplacePattern** methods to get started.

```
Document ActiveDoc = _applicationObject.ActiveDocument;
TextDocument TextDoc = (TextDocument)ActiveDoc.Object("TextDocument");
if (TextDoc != null)
{
    TextDoc.MarkText("public void");    // Bookmark all the lines we are going to tweak.
    TextDoc.ReplacePattern("public void", "sub");
}
```

In this simple example, we've searched for all C# void methods and converted them to sub calls (the VB equivalent). We've also marked the line we've changed so the user can navigate to the bookmarked lines to review the code changes.

Summary

The document and text document objects can perform some basic file manipulations and global text updates, but in the next two chapters we explore code modification in more detail, including the built-in Visual Studio code-parser class, the **code model**.

Chapter 12 Code Window

One of the common features that add-in modules offer is the ability to manipulate the code in windows. In this chapter, we will create an add-in to interact with the code in an open document window. We will explore how to pull code from the window, manipulate it, and write it back.

Simple code manipulation

We are still going to use the wizard to create our basic add-in, but rather than attach our code to the **Tools** menu, this time we will attach it to the context menu of the code window. So let's start up the wizard with the following:

- Visual C# (or your preferred language).
- Application Host: Only Visual Studio.
- Name/Description: *CodeHelp* and *Send sample of code to a guru...*
- Create UI Menu and do not load at start-up.

What this add-in will do is allow you to send an email with a sample of code to your local guru (hopefully you have one) and ask him or her what the code is doing. The add-in will then leave a [TODO] comment indicating that the code was sent to a guru and we need to document what is being accomplished by this code when the guru replies.

Attaching to the code window

Since our add-in module is going to attach to the code window instead of the main menu, we need to code our connection logic slightly differently.

```
object []contextGUIDS = new object[] { };
Commands2 commands = (Commands2)_applicationObject.Commands;
// Create the command object.
Command cmd = commands.AddNamedCommand2(_addInInstance, "CodeHelp", "CodeHelp",
    "Send sample to Guru..", true, 59, ref contextGUIDS,
    (int)vsCommandStatus.vsCommandStatusSupported +
    (int)vsCommandStatus.vsCommandStatusEnabled,
    (int)vsCommandStyle.vsCommandStylePictAndText,
    vsCommandControlType.vsCommandControlTypeButton);
// Create a command bar on the code window.
CommandBar CmdBar = ((CommandBars)_applicationObject.CommandBars)["Code Window"];

// Add a command to the Code Window's shortcut menu.
CommandBarControl cmdBarCtl = (CommandBarControl)cmd.AddControl(CmdBar,
    CmdBar.Controls.Count + 1);
cmdBarCtl.Caption = "Send sample to Guru..";
```

We are searching for the “Code Window” (precise spelling is important) and adding a pop-up menu control to it, rather than to the usual menu.

Responding to the click

When the user selects the menu item, the **Exec()** method will be called to process the **Send code to the Guru...** menu option.

Since the code will only be called from a code window (since our menu item is attached to the context menu), we can assume an active document will always be available.

Getting selected code

Once we have the selected text and confirmed some text was selected:

```
// See if any selected text.
Document theDoc = _applicationObject.ActiveDocument;
TextSelection sel = (TextSelection)theDoc.Selection;
if (sel.Text == "")
{
    MessageBox.Show("Please select some text...", "Error");
    handled = true;
    return;
}
```

We can build a mail message and send it to our guru.

```
// Let's make an e-mail for the guru.
string subjectLine = "Having trouble understanding this code";
string msgbody = "<p>Can you review it and tell me what the #@$* it is doing<p>" +
    Environment.NewLine + Environment.NewLine +
    "<pre>" + sel.Text + "</pre>" +
    Environment.NewLine + "<p>Thanks...";

MailMessage mail = new MailMessage();
mail.To.Add(GuruEmail);
mail.From = new MailAddress("xxxx@" + fromDomain);
mail.Subject = subjectLine;
mail.Body = msgbody;
mail.IsBodyHtml = true;
SmtpClient smtp = new SmtpClient();
smtp.Host = "smtp.gmail.com"; //Or your SMTP server address.
smtp.Credentials = new System.Net.NetworkCredential
    ("xxxxxxx@gmail.com", "password");
smtp.EnableSsl = true;
bool MailSent = false;
try
{
    Cursor.Current = Cursors.WaitCursor;
```

```

        smtp.Send(mail);
        MailSent = true;
    }
    catch
    {
        MessageBox.Show("Error sending mail", "ERROR",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
    Cursor.Current = Cursors.Default;

```



Note: You will need to add a reference to *System.net.mail* and define a string constant *GuruEmail* and a string constant *fromDomain* with your e-mail domain.

Tweak the code fragment

Now that we've sent our code to the guru for his or her comments and review, we want to mark the code with a to-do comment so we can add documentation when the guru replies to our email.

```

// Now we add the comment back to the code.
string commentChar = "//";
string the_TODOText =
    _applicationObject.ToolWindows.TaskList.DefaultCommentToken.ToString();
string revisedCode = commentChar + " " + the_TODOText + " GURU - sent to guru on " +
    DateTime.Now.ToString() + Environment.NewLine +
    commentChar + " < Guru answer here >" + Environment.NewLine +
    sel.Text +
    commentChar + " *****" + Environment.NewLine;
if (MailSent == false)
{
    revisedCode = commentChar + " " + the_TODOText + " Ask guru about this"+
        Environment.NewLine+
        sel.Text + Environment.NewLine+
        commentChar + " *****" + Environment.NewLine;
}

```

In this example, we are adding the comment and the to-do token that the IDE currently uses for tasks. This will allow our comment to be found easily using the task list window within Visual Studio.

Putting the code back

And now that we've assembled our revised code string, we need to update the editor with the revision. This is accomplished by copying the revised text to the Windows clipboard as text, and then pasting that text back to the editor.


```

DataObject theObj = new System.Windows.Forms.DataObject();
try
{
    theObj.SetData(System.Windows.Forms.DataFormats.Text, revisedCode);
    System.Windows.Forms.Clipboard.SetDataObject(theObj);
    sel.Paste();
}
catch
{
    MessageBox.Show("couldn't paste comment, sorry");
}

```

This is just a simple example of basic interaction with the contents of the code window. We don't perform any analysis of the content. We simply grab the code, tweak it somehow, and then put it back. However, this is only part of the capabilities built into Visual Studio.

Moving the code around

You do not have to put the samples or code revisions back where they came from. You can add code to the top or beginning of the file, or even at a random spot in the middle (although users are not likely to appreciate that).

Text Document

In order to manipulate the document, you need access to the **TextDocument** object associated with the current document. This is accomplished with the following command:

```

Document theDoc = _applicationObject.ActiveDocument;
TextDocument theText = (TextDocument)theDoc.Object();

```

The TextDocument object provides two point objects (**StartPoint** and **EndPoint**) referring to the expected locations in the body of text. In addition, there are some simple methods to mark and replace text within the document that matches a particular pattern.

For example, the following code sample will replace all occurrences of double with float in the document associated with the text document.

```

theText.ReplacePattern("double", "float");

```

You can also add FindOptions as the third parameter to control case matching, where to start searching the document, whether to match the whole word, etc. If you need to perform simple replacements on the entire document, the Text Document object provides the methods to do just that.

Edit point

The Text Document object also allows you to create an **EditPoint** object, which can be used to position your code anywhere within the document and make edits, deletions, insertions, etc. To create the edit point variable, use the following command:

```
EditPoint thePoint = theText.CreateEditPoint();
```

This allows a much finer degree of control over text manipulation. An edit point is the location in the file where you want to manipulate text. If you create an EditPoint object with no parameters, it is the same as the starting point from the text document. You can also specify a point as a parameter, so you could create an edit point based on the last location in the document by using EndPoint.

Once you have the EditPoint object, you have a variety of navigation functions to move through the text, such as:

- CharLeft(n): Move to the left *n* characters.
- CharRight(n): Move to the right *n* characters.
- EndOfLine(): Move to the end of the current line.
- LineDown(n): Move down *n* lines.
- LineUp(n): Move up *n* lines.
- WordLeft(n): Move to the left *n* words.
- WordRight(n): Move to the right *n* words.

When the point is positioned, you can insert text into the document. You can also use the **Get** methods to extract text from the document. For example, the following sample looks for lines beginning with **using** and adds a comment indicating the line needs to be converted to **imports** when converting from C# to Visual Basic.

```
EditPoint thePoint = theText.CreateEditPoint();
for (int x = 1; x < theText.EndPoint.Line; x++)
{
    if (thePoint.GetText(5).ToString() == "using")
    {
        thePoint.EndOfLine();
        thePoint.Insert(" // convert to imports statement");
    }
    thePoint.LineDown(1);
}
```

```
}
```

More complex code manipulation

While the methods and examples in this chapter allow simple text manipulations, they do not provide much in the way of parsing your source code. Fortunately, Visual Studio has a built-in class system that allows us to analyze code windows much more efficiently, without writing our own code parsers or worrying about VB.NET or C#. This system is the code model, and is described in the next chapter.

Chapter 13 Code Model

The code model is a language-independent view of a source code file. You can use this view to extract code elements from the classes found within a namespace down to the variables and methods within a class.

Using the code model

In order to illustrate how the code model system works, let's build a very simple class source file.

```
using System;

public class ComputePayrollAmount
{
    public string PersonName;
    public double payRate;
    private double TaxRate = 0.28F;           // Internally used in class.

    public void GetWeeklyPay()
    {
        double TotalPay = 40.0 * payRate;
        string checkLine = WriteCheck(PersonName, TotalPay);
    }
    private string WriteCheck(string forWhom, double Amount)
    {
        string result = "Pay to the order of " + forWhom + " $" + Amount.ToString();
        return result;
    }
}
```

You can access the code model for the active document using the following code sample:

```
FileCodeModel fileCM = dte.ActiveDocument.ProjectItem.FileCodeModel;
```

Once you have the code model available, one of the properties is a variable called **code elements**, which contains the code pieces at the current level. In our previous example, two code elements are returned:

- Import element.
- Class element.

The **import** element occurs once for each **import** or **using** statement in the file. Whether you are using VB, C++, or C#, each statement to import a module is included.

The second code element is the **class** statement and will contain the full name of the class, as well as an object property called **Children**. This object contains the code elements within the class structure. In this case, there will be five elements. The three variables and two methods are included in the Children object.

The fifth child of the class element refers to the **WriteCheck()** method, and it has two children elements, representing the parameters in the function.

Through recursive calls, you can easily trace a source file from its namespace, to classes within the file, to methods within the classes, to parameters of the methods.

The code element object can represent variables, methods, parameters, namespaces, etc. It has a **kind** property to know what you are working with, and point properties to keep track of location with the source file.

Get the code model of a source file

The code model is associated with a project item (see [Chapter 9](#)). Every document object that is part of a project has a **ProjectItem** property associated with it. Once you have a reference to the document, you can get the code model by using the following code. In this sample, we are getting the **CodeModel** property of the currently active document:

```
FileCodeModel fileCM = dte.ActiveDocument.ProjectItem.FileCodeModel;
```

The **FileCodeModel** class has two properties of interest; one is the language property that contains a GUID string indicating the type of language, C#, C++, or VB. You can use the following constants to determine what language the module is written in:

- vsCMLanguageCSharp
- vsCMLanguageVC
- vsCMLanguageVB

The other property is the collection of code elements. Each code element in this list can have a child collection of additional associated code elements, and can contain children elements as deep as the source code structure goes. Each item in the collection represents a single code element from the source file.

Code element properties

The key properties for working with individual code elements are listed in the following table:

Property	Data Type	Description
Children	Collection	Collection of nested code elements, if any.
FullName	String	Fully qualified (class and variable name) name of element.

Property	Data Type	Description
Kind	Enumerated	The type of element, such as: vsCMElementVariable, vsCMElementClass, etc.
Name	String	Name of the code element.
StartPoint	TextPoint	An object pointing to the beginning of the element.
EndPoint	TextPoint	An object referring to the end of the code element.

With these key properties, you can determine the type of code element you are working with, and you can extract it or write it back to the source code window using the point properties.

In addition to providing code elements, the file code model also allows you to add variable elements (classes, variables, namespaces, etc.), get a code element at a particular point in the code, and remove a code element from the source file.

Putting it all together

For our example project, we are going to create an add-in that will read a source code file and document the class details, as well as all the public variables and methods within the class.

To look at what the code will do, consider the following class example, before and after:

```
using System;

public class ComputePayrollAmount
{
    public string PersonName;
    public double payRate;

    enum Payclass
    {
        FullTime    = 1,
        PartTime    = 2,
        Consultants = 4
    };

    private string _firstName;
    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }

    private double TaxRate    = 0.28F;           // Internally used in class.

    private string _LastName;
    public string LastName
    {
```

```

        get { return _LastName; }
        set { _LastName = value; }
    }

    enum MaritalStatus
    {
        Single = 1,
        Married = 2,
        Seperated = 4
    };

    public void GetWeeklyPay()
    {
        double TotalPay = 40.0 * payRate;
        string checkLine = WriteCheck(PersonName, TotalPay);
    }
    public ComputePayrollAmount()
    {
    }

    private string WriteCheck(string forWhom, double Amount)
    {
        string result = "Pay to the order of " + forWhom + " $" + Amount.ToString();
        return result;
    }
}

```

This is often how classes built over time and by multiple developers end up: public variables, methods, properties, etc., all intermixed in the source code file. After running our add-in, the following comment code is generated and added to the top of the class definition:

```

// [=====]
//          Class: ComputePayrollAmount
//          Author: jbooth
//          Date: 10/4/2012
//
// Class Information:
//          Inherits: Object
//
// Public Interface:
//
//          Variables: PersonName (string)
//                   payRate (double) [40.0F]
//          Properties: FirstName (string)
//                   LastName (string)
//          Methods:  GetWeeklyPay
//                   WriteCheck(forWhom:string,Amount:double) ==> string
// [=====]

```

You can see in this example that the code model distinguished between public and private variables and methods, and also discerned enough to not include the constructor in the list of public methods.

For simplicity's sake, our code assumes a single class in a source file, and will only process the first class it finds. However, you can use this concept as a starting point for enforcing coding standards, making code more readable, etc.



Note: The code will overwrite the existing comment if you run it multiple times.

Class documenter

We are still going to use the wizard to create our basic add-in, and then attach our module to the context menu of the code window. So let's start up the wizard with the following:

- Visual C# (or your preferred language).
- Application Host: Only Visual Studio.
- Name/Description: *DocumentClass* and *Document a class file*
- Create UI Menu and do not load at start-up.

Attaching to the code editor window

The first change we want to make is to move the menu option to appear on the **Code window context** menu rather than the **Tools** menu. Our connect code should be changed to the following:

```
if(connectMode == ext_ConnectMode.ext_cm_UISetup)
{
    // Create the command object.
    object[] contextGUIDS = new object[] { };
    Commands2 commands = (Commands2)_applicationObject.Commands;
    try
    {
        Command cmd = commands.AddNamedCommand2(_addInInstance, "DocumentClass",
            "Class Documentator", "Document your class module ", true, 59, ref contextGUIDS,
            (int)vsCommandStatus.vsCommandStatusSupported +
            (int)vsCommandStatus.vsCommandStatusEnabled,
            (int)vsCommandStyle.vsCommandStylePictAndText,
            vsCommandControlType.vsCommandControlTypeButton);
        // Create a command bar on the code window.
        CommandBar CmdBar = ((CommandBars)_applicationObject.CommandBars)["Code Window"];
        // Add a command to the Code Window's shortcut menu.
        CommandBarControl cmdBarCtl = (CommandBarControl)cmd.AddControl(CmdBar,
                                                                    CmdBar.Controls.Count + 1);
        cmdBarCtl.Caption = "Class Doc";
    }
    catch (System.ArgumentException)
    {
    }
}
```


Getting the code model

We now need to add our code to the **Exec()** function to get the code model and use its parsing ability to create a documentation header.

```
if(commandName == "DocumentClass.Connect.DocumentClass")
{
    FileCodeModel2 fileCM = null;
    // Make sure there is an open source-code file.
    try
    {
        fileCM = (FileCodeModel2)_applicationObject.ActiveDocument.
                    ProjectItem.FileCodeModel;
    }
    catch
    {
        MessageBox.Show("No active source file is open...");
        handled = true;
        return;
    }
    // Some files (such as XML) will not have an associated code model.
    if (fileCM == null)
    {
        MessageBox.Show("Not a valid programming language source file...");
        handled = true;
        return;
    }
}
```

Assuming we've gotten a code model (valid source file), we need to make sure it is a language we can work with, in this case, either VB or C#.

```
string CommentChar = "";
switch (fileCM.Language)
{
    case CodeModelLanguageConstants.vsCMLanguageCSharp:
    {
        CommentChar = "//";
        break;
    }
    case CodeModelLanguageConstants.vsCMLanguageVB:
    {
        CommentChar = "'";
        break;
    }
}

if (CommentChar == "")
{
    MessageBox.Show("Only works with VB or C# class modules");
    handled = true;
    return;
}
```

Finding the class elements

Once we know we've got a valid code module, we need to find the code element to locate the first class. Typically, a class declaration is either in the first level, or one level down (child level of the Namespace level). The following code will search two levels deep for a class code element:

```
// Scan the file, looking for a class construct may require two passes.
CodeElements elts = fileCM.CodeElements;
CodeElement elt = null;
int xClassElt = 0;
int xNameSpace = 0;
int nLevels = 0;
while (xClassElt == 0)
{
    nLevels++;
    for (int i = 1; i <= elts.Count; i++)
    {
        elt = elts.Item(i);
        if (elt.Kind == vsCMElement.vsCMElementClass)
        {
            xClassElt = i;
            break;
        }
        if (elt.Kind == vsCMElement.vsCMElementNamespace)
        {
            xNameSpace = i;
            break;
        }
    }
    // Found namespace and no class, let's work through the namespace looking for a class.
    if (xNameSpace != 0 && xClassElt == 0)
    {
        elts = elts.Item(xNameSpace).Children;
    }
    // Don't search deeper than three levels.
    if (nLevels > 2) { break; }
}
// If no class found, exit.
if (xClassElt == 0)
{
    MessageBox.Show("No class module found in source file...");
    handled = true;
    return;
}
```

Once we've found a class element, we grab the child elements (i.e. the variables, methods, etc. that we want to document.)

```
// Now we are ready to document our class.
CodeClass theclass = (CodeClass)elts.Item(xClassElt);
```

```
object[] interfaces = {};  
object[] bases = {};
```

Notice that we've cast the generic **CodeElement** to the more specific **CodeClass** object, which gives us access to the particular class details. This type casting is necessary to pull additional information from the various code elements, rather than just relying on the properties in the generic **CodeElement** class.

Building our header

We now create a string builder object and extract some information from our class variable to display in the documentation header text.

```
// Some initial header info.  
StringBuilder sb = new StringBuilder();  
sb.AppendLine(CommentChar+"[=====]");  
if (theclass.Namespace != null)  
{  
    sb.AppendLine(CommentChar + "        Namespace: " +  
                  theclass.Namespace.Name.ToString());  
}  
if (theclass.IsAbstract)  
{  
    sb.Append( CommentChar+"    Abstract ");  
}  
else  
{  
    sb.Append( CommentChar+"    ");  
}  
sb.AppendLine("Class: "+theclass.Name);  
sb.AppendLine( CommentChar+"        Author: "+Environment.UserName);  
sb.AppendLine( CommentChar+"        Date: "+DateTime.Now.ToShortDateString());  
sb.AppendLine( CommentChar+"    ");  
sb.AppendLine( CommentChar+"    Class Information:");  
// Information about the class.  
string docCategory = "        Inherits:";  
foreach (CodeElement theBase in theclass.Bases)  
{  
    sb.AppendLine(CommentChar + docCategory + " " + theBase.Name);  
    docCategory = "        ";  
}  
docCategory = "        Implements:";  
foreach (CodeElement theImpl in theclass.ImplementedInterfaces)  
{  
    sb.AppendLine(CommentChar + docCategory + " " + theImpl.Name);  
    docCategory = "        ";  
}  
sb.AppendLine( CommentChar+"    ");  
sb.AppendLine( CommentChar+"    Public Interface:");  
sb.AppendLine( CommentChar+"    ");
```

You can look to the **CodeClass** object for more information you might want to display. Our next step is to collect the code elements making up the class and store them in a collection so that we can display them grouped by element type later in the module.

Organizing the code elements

This next section of the code loops through the class elements and organizes them by type into queue structures. Note that we are testing the generic code element's type and then storing the appropriate typed element type (i.e. `CodeVariable`, `CodeEnum`, etc.) into the queue structure for the particular element kind.

```
elts = theclass.Children;
// Build queues to hold various elements.
Queue<CodeEnum> EnumStack = new Queue<CodeEnum>();
Queue<CodeVariable> VarStack = new Queue<CodeVariable>();
Queue<CodeProperty> PropStack = new Queue<CodeProperty>();
Queue<CodeFunction> FuncStack = new Queue<CodeFunction>();

foreach (CodeElement oneElt in elts)
{
    // Get the code element, and determine its type.
    switch (oneElt.Kind)
    {
        case vsCMElement.vsCMElementEnum :
        {
            EnumStack.Enqueue((CodeEnum)oneElt);
            break;
        }
        case vsCMElement.vsCMElementVariable:
        {
            VarStack.Enqueue((CodeVariable)oneElt);
            break;
        }
        case vsCMElement.vsCMElementProperty:
        {
            PropStack.Enqueue((CodeProperty)oneElt);
            break;
        }
        case vsCMElement.vsCMElementFunction:
        {
            FuncStack.Enqueue((CodeFunction)oneElt);
            break;
        }
        default : { break; };
    }
}
```

Once we've collected and built our queue collections, we can now write them back to the documentation comment text organized by code element type.

Variables

For each public variable, we want to report the variable name and data type, as well as any initial value it might be set to. Note that the variables (and all code elements) are documented in the order they are encountered; they are not sorted in the queue.

```
// Iterate through the variables looking for public variables.
foreach (CodeVariable theVar in VarStack)
{
    if (theVar.Access == vsCMAccess.vsCMAccessPublic)
    {
        sb.Append(CommentChar + docCategory + " " + theVar.Name +
            " (" + theVar.Type.AsString + ")");
        docCategory = " ";
        if (!(theVar.InitExpression == null))
        {
            sb.Append(" [" + theVar.InitExpression.ToString() + "]");
        }
        sb.AppendLine("");
    }
}
```

Enums

For the enumerated types, we want to report the name of the enumeration and all of the elements (stored as children variables to the enum itself). The following code illustrates how to walk through the enum and its children elements.

```
docCategory = "          Enums:";
foreach (CodeEnum theEnum in EnumStack)
{
    if (theEnum.Access == vsCMAccess.vsCMAccessPublic)
    {
        sb.Append(CommentChar + docCategory + " " + theEnum.Name + " ");
        docCategory = " ";
        if (theEnum.Children.Count > 0)
        {
            sb.Append("(");
            for (int xx = 1; xx <= theEnum.Children.Count; xx++)
            {
                int yy = theEnum.Children.Count - xx + 1;
                CodeVariable theVar = (CodeVariable)theEnum.Children.Item(yy);
                sb.Append(theVar.Name);
                if (yy > 1) { sb.Append(","); }
            }
            sb.Append(")");
            sb.AppendLine("");
        }
    }
}
```

Properties

For each public property, we want to display the property name and the property's data type. The following code loops through the collected **CodeProperty** elements and does just that.

```
docCategory = "          Properties:";
foreach (CodeProperty theProp in PropStack)
{
    if (theProp.Access == vsCMAccess.vsCMAccessPublic)
    {
        sb.Append(CommentChar + docCategory + " " + theProp.Name +
                  " (" + theProp.Type.AsString + ")");
        sb.AppendLine("");
        docCategory = "          ";
    }
}
```

Methods

As we process the class methods, we want to only report on public methods and exclude the constructor from the documentation. We need to handle parameters and the return type (if not VOID).

```
docCategory = "          Methods:";
foreach (CodeFunction theFunc in FuncStack)
{
    if (theFunc.FunctionKind != vsCMFunction.vsCMFunctionConstructor &&
        theFunc.Access==vsCMAccess.vsCMAccessPublic)
    {
        sb.Append(CommentChar + docCategory + " " + theFunc.Name);
        docCategory = "          ";
        if (theFunc.Parameters.Count > 0)
        {
            int yy = theFunc.Parameters.Count;
            sb.Append("(");
            foreach (CodeParameter theParam in theFunc.Parameters)
            {
                sb.Append(theParam.Name+": "+theParam.Type.AsString);
                yy--;
                if (yy > 0) { sb.Append(","); }
            }
            sb.Append(")");
        }
        if (theFunc.Type.AsString.ToUpper().EndsWith("VOID")==false)
        {
            sb.Append(" ==> " + theFunc.Type.AsString);
        }
        sb.AppendLine("");
    }
}
```

Writing the header back to the source window

By this point, we have a nicely formatted documentation block of text, showing all the public elements of the class. We are going to use our text document and edit point objects to either write the text to the top of the file, or update the prior version of the documentation. This allows you to run the add-in as often as you want after you've added new public code elements to the class.

```
TextDocument theText = (TextDocument)_applicationObject.ActiveDocument.Object();
EditPoint thePoint = theText.CreateEditPoint();
// Check and see if a comment already exists.
string theLine = thePoint.GetText(thePoint.LineLength);
bool FoundOldComment = false;
string OldComment = theLine+Environment.NewLine;

if (theLine.StartsWith(CommentChar + " [==="))           // Start of delimiter.
{
    while (thePoint.AtEndOfDocument == false && FoundOldComment==false)
    {
        thePoint.LineDown(1);
        theLine = thePoint.GetText(thePoint.LineLength);
        OldComment+= theLine+Environment.NewLine;
        FoundOldComment = theLine.StartsWith(CommentChar) &&
                           theLine.EndsWith("==]");
    }
}
if (FoundOldComment)
{
    thePoint = theText.CreateEditPoint(theText.StartPoint);
    thePoint.ReplacePattern(theText.EndPoint, OldComment, sb.ToString());
}
else
{
    thePoint.Insert(sb.ToString());
}
```

Summary

The code model features of Visual Studio allow you to determine code elements without the need to write your own parsing routines. Although not all elements are returned in the code collection (such as compiler directives), the model gives you a great starting point for writing add-in modules to work with the code in a source file.

Chapter 14 Tool Windows

The Visual Studio IDE consists of a number of different tool windows to manage the solution. You can access these windows through the **ToolWindows** property of your **_applicationObject** variable. A few of the commonly used windows have classes written specifically for those windows, but every window is accessible, either through one of the common classes or through the **GetToolWindow()** method.

Error List

The Error List window contains all errors, warnings, and messages that the most recent compile or build step encountered.

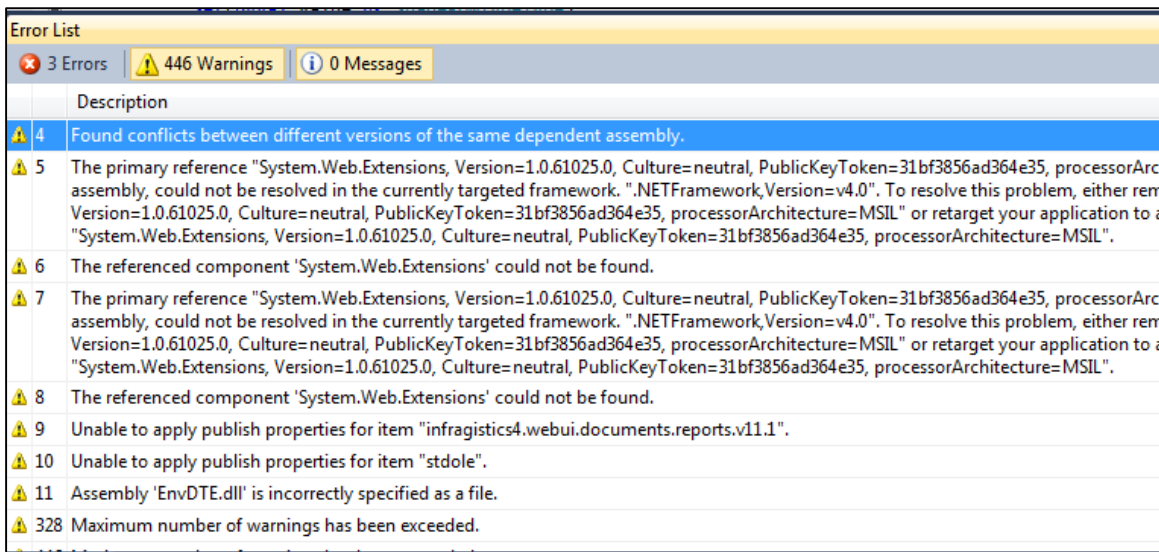


Figure 13: Error List

You can programmatically access the error messages using the Error List window.

```
EnvDTE80.ErrorList errList = _applicationObject.ToolWindows.ErrorList;
```

The Error List object contains three Boolean properties, indicating which messages are included in the error list:

- ShowErrors
- ShowMessages
- ShowWarnings

You can toggle these properties to control the content of the error items list.

Task List

Visual Studio provides a Task List Manager which allows developers to build a task list by entering tasks or by adding TODO comments in the source code.

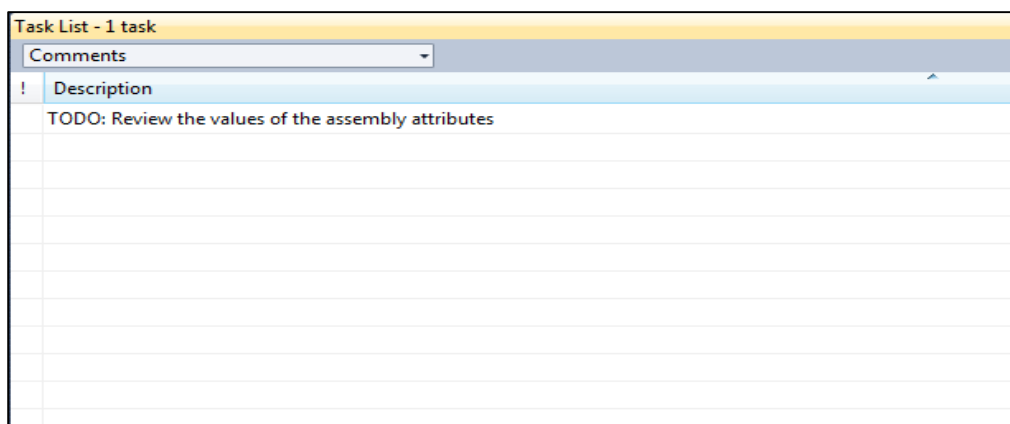


Figure 14: Task List Manager

You can programmatically access the **Task List** window and all tasks with your add-in using the following code:

```
EnvDTE.TaskList TaskList = _applicationObject.ToolWindows.TaskList;
```

The task list object allows you to read or set the **Default Comment Token** (which is usually “TODO”) via the **DefaultCommentToken** string property.

The primary interface with the task list is the **TaskItems** object. You can find the number of items in the list using the integer **Count** property. You can also get details about any single task by using the **Item()** indexed property. This will return an individual task item entry, with the following properties:

Property	Data Type	Description
Category	String	Comment or User Task
Checked	Boolean	Is the task item checked?
Description	String	Descriptive text of the task
Displayed	Boolean	Is the task item currently displayed?
FileName	String	If a file is associated with the task, its fully qualified path name is provided

Property	Data Type	Description
Line	Integer	Line number in file where TODO comment is found
Priority	Enum	vsTaskPriorityLow, PriorityMedium, PriorityHigh

Solution Explorer

The **Solution Explorer** window shows a tree view UI element displaying the currently open solution, as the following figure illustrates:

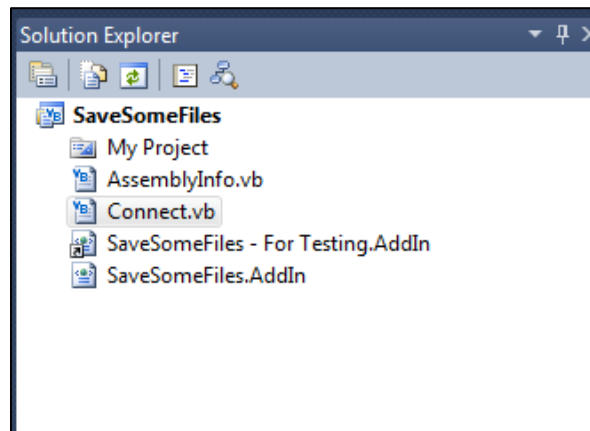


Figure 15: Solution Explorer

You can access the **Solution Explorer** using the following code:

```
EnvDTE.UIHierarchy SolExplore = _applicationObject.ToolWindows.SolutionExplorer
```

You can traverse the tree of the **Solution Explorer** using the **UIHierarchyItems** property to provide you access to each level of the tree display. Each item represents a single element in the view. In our previous example, the first hierarchy item would be the **SaveSomeFiles** level. That item would have a **UIHierarchyItems** collection as well, which would contain the **My Project** item, the **AssemblyInfo.vb** item, **Connect.vb**, etc.

Output Window

The **Output Window** is a text window showing the output of various IDE tools, such as the build process, the debug process, etc. You can use the **ToolWindow** object to gain access to the **Output Window**, as shown in the following example:

```
OutputWindow outWnd = _applicationObject.ToolWindows.OutputWindow;
```

Each tool has its own pane, which is selected by the user via a drop-down menu. You can add your own pane if you want a place to collect and display messages from your own tool. For example, the following code adds a pane to keep track of user interface issues your tool might discover:

```
OutputWindowPane OutputPane = outWnd.OutputWindowPanes.Add("UI issue ");
```

Searching for bad words

Another useful add-in module we could write would search a solution's source projects looking for a list of "forbidden words." A famous CAD design software application once contained a message (probably left over by a programmer) that said, "this is a message, you idiot." While the programmer might have thought it was humorous, the company that had to write an apology letter and send out a patched version of the software probably didn't see the humor. In order to prevent a similar incident, we can write an add-in to search all files within a project and add to the error list any occurrences of the forbidden words. I'll define that as a regular expression constant so you can create your own list of words.

For this add-in, we will add a button to the standard toolbar. If a solution is open, we will scan all the projects and add the bad words and locations to our own pane in the output window. We will also add an entry into the task list with a high priority to clean the words up.

Bad words scan

We are still going to use the wizard to create our basic add-in, and then attach our module to the standard toolbar of the IDE. So let's start up the wizard with the following:

- Visual C# (or your preferred language).
- Application Host: Only Visual Studio.
- Name/Description: *Bad Words* and *Scan files for bad words*.
- Create UI Menu and do not load at start-up.

After the wizard generates your class source file, add the following variables to the class definition. You can customize your list of words by adding them to the BAD_WORD_LIST string.

```
const int RED_STAR_ICON = 6743;  
const string BAD_WORD_LIST = "(stupid|idiot|fool)";  
bool AddedToTaskList = false;
```

Using a tool button

For this example, we are going to call our add-in module from the toolbar rather than a menu item. We will also disable the item if a solution is not open in Visual Studio, since this add-in searches all project items with a solution in order to mark the occurrences of your bad word list. Change your **OnConnection** method code to add an icon to the standard toolbar instead, as shown in the following code sample:

```
// Add the command.
Command cmd = (Command)_applicationObject.Commands.AddNamedCommand(_addInInstance,
    "BadWords", "BadWords",
    "Search for bad words", true, RED_STAR_ICON, null,
    (int)vsCommandStatus.vsCommandStatusSupported +
    (int)vsCommandStatus.vsCommandStatusEnabled);
CommandBar stdCmdBar = null;
// Reference the Visual Studio standard toolbar.
CommandBars commandBars = (CommandBars)_applicationObject.CommandBars;
foreach (CommandBar cb in commandBars)
{
    if(cb.Name=="Standard")
    { stdCmdBar = cb;
      break; }
}
// Add a button to the standard toolbar.
CommandBarControl stdCmdBarCtl = (CommandBarControl)cmd.AddControl(stdCmdBar,
    stdCmdBar.Controls.Count + 1);
// Set a caption for the toolbar button.
stdCmdBarCtl.Caption = "Search for bad words";
// Set the toolbar's button style to an icon button.
CommandBarButton cmdBarBtn = (CommandBarButton)stdCmdBarCtl;
cmdBarBtn.Style = MsoButtonStyle.msoButtonIcon;
```

Only if a solution is open

Since we only want to display the icon if a solution is open, we need to add code to check that condition during our **QueryStatus** method call.

```
if(commandName == "BadWords.Connect.BadWords")
{
    if (_applicationObject.Solution.Count > 0)
    {
        status = (vsCommandStatus)vsCommandStatus.vsCommandStatusSupported |
            vsCommandStatus.vsCommandStatusEnabled;
    }
    return;
}
```

Getting tool windows

In our **Exec()** method, we want to make sure a solution is open and if so, get references to the **Output** window and the task list. We also want to add our custom pane called “Bad Words” and activate the **Output** window, as shown in the following code sample:

```
handled = true;
if (_applicationObject.Solution.Count < 1)
{
    MessageBox.Show("Please open a solution to scan...");
    return;
}
// Need to get all project items and search for "bad words".
OutputWindow outWnd = _applicationObject.ToolWindows.OutputWindow;
TaskList theTasks = _applicationObject.ToolWindows.TaskList;
OutputWindowPane OutputPane = outWnd.OutputWindowPanes.Add("Bad words");
OutputPane.Clear();
bool FoundBadWords = false;
// Activate the output window.
Window win = _applicationObject.Windows.Item(EnvDTE.Constants.vsWindowKindOutput);
win.Activate();
```

Looping through the project

We are now ready to loop through the solution and all projects within. Within each project, we search through the project items. If the project item has a document object attached to it, and the document item contains a text document object, we’ve found a file with text (source, XML configuration file, etc.) that we should scan for entries. Some files, such as the Assembly file, will not have a text document object, so we will skip scanning those files.

```
foreach (Project CurProject in _applicationObject.Solution)
{
    foreach (ProjectItem CurItem in CurProject.ProjectItems)
    {
        Document theDoc = null;
        Try
        { theDoc = CurItem.Document; }
        catch
        { }
        if (theDoc != null)
        {
            TextDocument theText = (TextDocument)theDoc.Object("TextDocument");
            if (theText != null)
            {

```

Marking bad words

Using the **Mark Text** method of the Text Document object, we apply a regular expression search to see if the file contains any words from the list. The lines are bookmarked and the file name is added to our **Output** window. The following code performs the search task:

```

if (theText.MarkText(BAD_WORD_LIST, (int)vsFindOptions.vsFindOptionsRegularExpression))
{
    OutputPane.OutputString(CurItem.Name + " contains bad words"+Environment.NewLine);
    FoundBadWords = true;
}

```

Adding a clean-up task

Our final step in the process is to add an entry to the task list, reminding the programmer to clean up the code that contains the bad words. We only do this if the add-in found bad words and we've not yet added the task. The task priority and task icon control the appearance of the task in the task, with red being a high priority item.

```

if (FoundBadWords && AddedToTaskList==false)
{
    TaskItems2 TLItems = (TaskItems2)theTasks.TaskItems;
    TLItems.Add("Bad Words", "Bad Words", "Remove bad words " + BAD_WORD_LIST +
        " from source files",
        vsTaskPriority.vsTaskPriorityHigh, vsTaskIcon.vsTaskIconNone,
        true, null, 10, true, true);
    AddedToTaskList = true;
}

```

When the add-in completes, it will have marked all lines with words from your bad word list and added the list of files to the output window with the drop-down pane of bad words.

Summary

While the **Tool** windows have a more narrow focus than the general source editing windows, the object types available provide your add-in with the ability to integrate easily with the toolbars, so you can add your custom output, save to-do items, etc.

Chapter 15 Source Code Generation

One time-saving option you can add to the Visual Studio IDE is the ability to generate source code. As developers, there are often common code-writing tasks we need to perform, and by designing an input screen and code generator, we can create an add-in module to save time with the development cycle.

Source code helper class

To assist in the generation of source code, it can be beneficial to create a helper class, which is basically a collection of routines to perform some common tasks that are likely to occur in generating code. We start our class definition by defining the different programming languages we want to support and providing a property to allow users to decide which language they want to generate code in.

```
// <summary>
// Code Gen class: Helper class to generate code for add-ins.
// </summary>
public class CodeGen
{
    // <summary>
    // List of programming languages generator works with.
    // </summary>
    public enum ProgrammingLanguages
    {
        VisualBasic = 1,
        CSharp = 2
    }

    private ProgrammingLanguages theLang = ProgrammingLanguages.VisualBasic;
    private bool inComment = false;

    // <summary>
    // Programming language to generate code for.
    // </summary>
    public ProgrammingLanguages Programming_Language {
        get { return theLang; }
        set { theLang = value; }
    }
}
```

Once the basic class is created, we can add some methods to generate the appropriate comment text. The simplest method is **SingleLineComment()** which generates the appropriate syntax and comment text for a comment in a line of code.

```
public string SingleLineComment(string theComment)
```

```

{
    string res = string.Empty;
    switch (theLang) {
        case ProgrammingLanguages.CSharp:
            res = "// " + theComment;
            break;
        case ProgrammingLanguages.VisualBasic:
            res = "' " + theComment;
            break;
    }
    return res;
}

```

The code determines the appropriate delimiter based on the chosen programming language, and then returns a string of the delimiter and the comment text. We can also add a method called **StartComment()**, which writes a multi-line comment starting delimiter and sets a flag to indicate we are in commented code.

```

public string StartComment()
{
    return StartComment("");
}

public string StartComment(string theComment)
{
    string res = string.Empty;
    switch (theLang) {
        case ProgrammingLanguages.CSharp:
            res = "/* " + theComment;
            break;
        case ProgrammingLanguages.VisualBasic:
            res = "' " + theComment;
            break;
    }
    inComment = true;
    return res;
}

```

The **StopComment()** method writes the appropriate ending comment delimiter and turns off the commenting flag.

```

public string StopComment()
{
    string res = string.Empty;
    switch (theLang) {
        case ProgrammingLanguages.CSharp:
            res = "*/ ";
            break;
        case ProgrammingLanguages.VisualBasic:
            break;
    }
}

```



```

    }
    inComment = false;
    return res;
}

```

There are a couple of additional methods to round out our helper class. These include **MakeFileName()**, which is used to append the appropriate extension to a file name.

```

public string MakeFileName(string theName)
{
    string res = theName;
    switch (theLang)
    {
        case ProgrammingLanguages.CSharp:
            res += ".cs";
            break;
        case ProgrammingLanguages.VisualBasic:
            res += ".vb";
            break;
    }
    return res;
}

```

We can also use **DeclareVariable()** to create a variable in any of the languages.

```

public string DeclareVariable(string varName, string DataType, string DefaultValue)
{
    string res = string.Empty;
    switch (theLang)
    {
        case ProgrammingLanguages.CSharp:
            res = DataType + " " + varName;
            if (DefaultValue.Length > 0)
            { res += " = " + DefaultValue; }
            res += ";";
            break;
        case ProgrammingLanguages.VisualBasic:
            res = "DIM " + varName + " AS " + DataType;
            if (DefaultValue.Length > 0)
            { res += " = " + DefaultValue; }
            break;
    }
    return res;
}

```

Our final function, **StartRoutine()**, returns a function declaration and delimiter shell.

```

public string StartRoutine(string typeOfCall, string RoutineName, string ReturnType)
{

```

```

string res = string.Empty;
switch (theLang)
{
    case ProgrammingLanguages.CSharp:
        if (typeOfCall.StartsWith("P"))
        { res = "public void "; }
        else
        { res = "public "+ReturnType+" "; }
        res += RoutineName+"()" + Environment.NewLine;
        res += "{";
        break;
    case ProgrammingLanguages.VisualBasic:
        if (typeOfCall.StartsWith("P"))
        { res = "sub " + RoutineName; }
        else
        { res = "function " + RoutineName + "()" as " + ReturnType; }
        res += Environment.NewLine;
        break;
}
return res;
}

```

With this simple class library available to help out code generation, we can now begin our add-in code.

Standardized headers

Imagine your company has a set of standard headers that every code module must include. These headers include the date and time the program was created, as well as the version of Visual Studio used to create the file.

Wizard settings

Start your standard headers add-in using the wizard and the following settings:

- Visual C# (or your preferred language).
- Application Host: Only Visual Studio.
- Name/Description: *StdHeaders* and *Generate a standard heading module*.
- Create UI Menu and do not load at start-up.

Verify the settings at the **Summary** screen, and if they look okay, generate the code.

Moving to File menu

For our standard headers add-in, we would rather have the menu item attached to the **File** menu using an icon instead of the **Tools** menu. We need to change a couple of lines in our **onConnection** method:

```
public void OnConnection(object application, ext_ConnectMode connectMode,
                        object addInInst, ref Array custom)
{
    _applicationObject = (DTE2)application;
    _addInInstance = (AddIn)addInInst;
    if(connectMode == ext_ConnectMode.ext_cm_UISetup)
    {
        object []contextGUIDS = new object[] { };
        Commands2 commands = (Commands2)_applicationObject.Commands;
        string toolsMenuName = "File";
    }
}
```

In this case, we are changing the **toolsMenuName** variable from **Tools** to **File**.

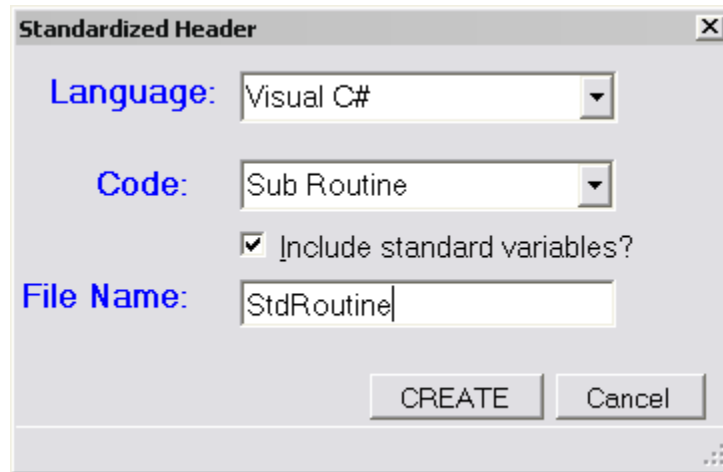
```
const int DOCUMENTS_ICON = 1197;
```

We will also add the constant for the documents icon, and use that value rather than the hard-coded 59 in the **AddNamedCommand2()** call.

```
//Add a command to the Commands collection:
Command command = commands.AddNamedCommand2(_addInInstance, "StdHeaders",
    "StdHeaders", "Standardized headers", true,
    DOCUMENTS_ICON,
    ref contextGUIDS,
    (int)vsCommandStatus.vsCommandStatusSupported+
    (int)vsCommandStatus.vsCommandStatusEnabled,
    (int)vsCommandStyle.vsCommandStylePictAndText,
    vsCommandControlType.vsCommandControlTypeButton);
```

Options screen

The options screen is a Windows form that asks users a few questions about the type of code header they want to generate. As a starting point, we will need to know the file to save, the programming language to use, and whether to generate a function or subroutine call.



The image shows a Windows dialog box titled "Standardized Header". It contains three main sections: "Language:" with a dropdown menu showing "Visual C#", "Code:" with a dropdown menu showing "Sub Routine", and "File Name:" with a text box containing "StdRoutine". There is a checkbox labeled "Include standard variables?" which is checked. At the bottom right are two buttons: "CREATE" and "Cancel".

Figure 16: Options screen for standard header add-in

Generate the header

If the user clicks **OK** to generate the header code, we will pull the selected options from the Windows form and use them to set our generated header and code template.

```
StringBuilder sb = new StringBuilder(); // String to hold header.
CodeGen Gen = new CodeGen(); // Code generation helper.
StdHeaderForm theForm = new StdHeaderForm();
theForm.ShowDialog();
if (theForm.DialogResult == DialogResult.OK)
{
    string cFile = theForm.CODEFILE.Text;
    // Get programming language choice.
    switch (theForm.LANGCOMBO.SelectedIndex)
    {
        case 0:
        { Gen.Programming_Language = CodeGen.ProgrammingLanguages.CSharp;
          break;
        }
        case 1:
        { Gen.Programming_Language = CodeGen.ProgrammingLanguages.VisualBasic;
          break;
        }
    }

    sb.AppendLine(Gen.StartComment());
    sb.AppendLine(Gen.WriteCode("====="));
    sb.AppendLine(Gen.WriteCode("    Program: " + cFile ));
    sb.AppendLine(Gen.WriteCode("    Author: " + Environment.UserName));
    sb.AppendLine(Gen.WriteCode("    Date/Time: " + DateTime.Now.ToShortDateString() +
        "/" + DateTime.Now.ToShortTimeString()));
    sb.AppendLine(Gen.WriteCode("    Environment: Visual Studio " +
        _applicationObject.Edition));
}
```

```
sb.AppendLine(Gen.WriteCode("====="));
sb.AppendLine(Gen.StopComment());
sb.AppendLine(Gen.WriteCode(""));
```

Add sub/function call

The following code sample adds a call to the routine TBD (hopefully the developer will update the name of the routine):

```
//Write the function prototype.
sb.AppendLine(Gen.StartRoutine(theForm.TYPECOMBO.Text.ToUpper(), "TBD", "string"));
```

Add standard variables

In some applications, standard variables are used so that every programmer uses the same variable names and meanings. In our example here, we use variables called **SourceModified** and **StartTime** to track modifications and monitor performance.

```
// Optionally, write standard variables.
if (theForm.INCLUDECHECK.Checked)
{
    sb.AppendLine(Gen.DeclareVariable("SourceModified", "string"));
    sb.AppendLine(Gen.DeclareVariable("StartTime", "DateTime", "DateTime.Now()"));
}
sb.AppendLine(Gen.EndRoutine(theForm.TYPECOMBO.Text.ToUpper()));
```

Open a new window

Once the string builder variable is created with the headers, we need to save it to a file and then open it within the IDE.

```
cFile = Gen.MakeFileName(cFile);
StreamWriter objWriter = new System.IO.StreamWriter(cfile);
objWriter.Write(sb.ToString());
objWriter.Close();

ItemOperations itemOp;
itemOp = _applicationObject.ItemOperations;
itemOp.OpenFile(cfile, Constants.vsViewKindCode);
```

After the add-in completes, a new window will be open with code similar to the following example:

```
'
' =====
'   Program: Sample
'   Author: Joe
'   Date/Time: 10/13/2012/9:24 AM
'   Environment: Visual Studio Professional
' =====
'
Function TBD() As String
    Dim SourceModified As String
    Dim StartTime As DateTime = DateTime.Now()
End Function
```

Item Operations object

The **ItemOperations** object of the **_applicationObject** provides methods to open and add files in the Visual Studio IDE. In the previous example, we've created the file, and using the **ItemOp** variable, instructed Visual Studio to open the file in a code editor window. The object allows you to programmatically perform some of the options from the **File** menu.

Other methods of the **Item Operations** object include:

- **AddExistingItem()**: Add an existing file to the project.
- **AddNewItem()**: Add a new item to the project. You can pass two parameters: the category name/item name (such as General/XML File), and the display name in the project window.
- **IsFileOpen()**: Is the file name passed as a parameter open in an IDE window?
- **Navigate()**: Open a browser window to a specified URL.
- **NewFile()**: Create a new file using the virtual path indicating the type of file. You can optionally specify the file name for the item and the view kind to open the file in.
- **OpenFile()**: Open an existing file in the editor using a specified view kind. In our example code, we created a file and opened it in a code view.

Summary

This chapter demonstrated how to build a source code file by pulling information from Visual Studio and the environment to create a standardized header. It also showed how to open the source file in a Visual Studio document window to allow the user to start programming immediately.

Chapter 16 Deploying Your Add-In

Once the add-in module you have developed is completed, debugged, and ready to go, you probably will want to share it with your fellow developers. In this chapter, we will cover what needs to be done to install your add-in and to interact with it through the Add-in Manager.

Installing the add-in

To install your add-in, you'll need to copy two files to one of the folders where Visual Studio looks for add-in modules. This is usually **\Documents\Visual Studio 2010\Addins** in Visual Studio 2010, or **\Documents\Visual Studio 2012\Addins** in Visual Studio 2012. You can also look in Visual Studio's **Options** dialog, under the **Environment** node's **Add-in/Macro Security** page for the **Add-in File Paths** list.

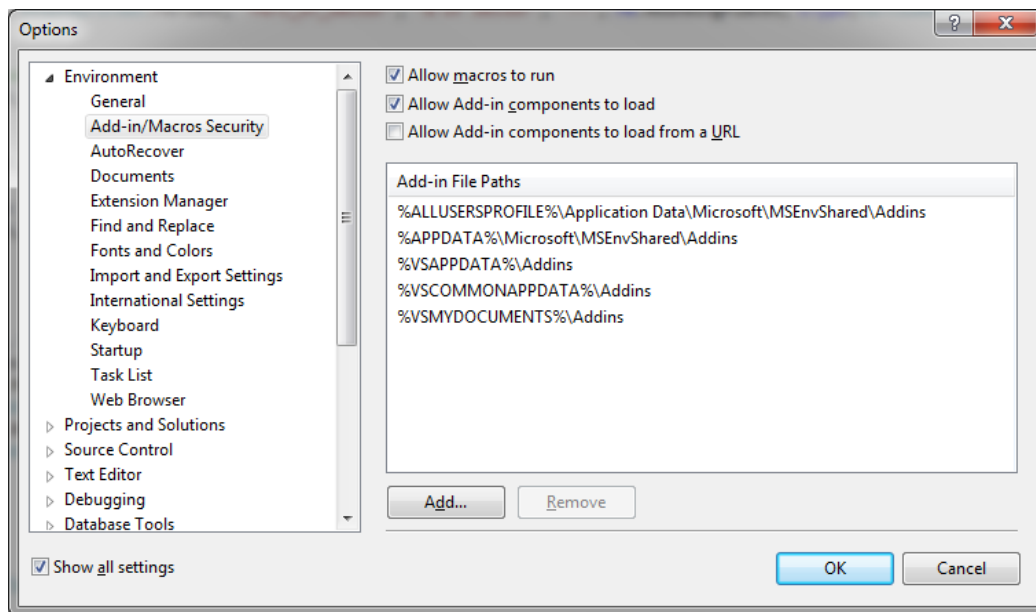


Figure 17: Installing add-ins



Tip: You might need to select **Show All Settings** if the **Add-in/Macro Security** page does not appear.

If you copy the Assembly DLL file and the .AddIn XML files to this folder, Visual Studio will discover it and possibly load it the next time Visual Studio is started. (The .AddIn XML files have options to control when the add-in is loaded. See [Chapter 6](#) for details.)

Add-in Manager

The **Add-in Manager** is a tool window under the **Tools** menu that lets you interact with all IDE-installed add-ins. You can change when the add-in is loaded, and you can disable the add-in as well. The descriptive text you've been entering in various add-in modules in this book will appear in the **Add-in Manager** tool window.

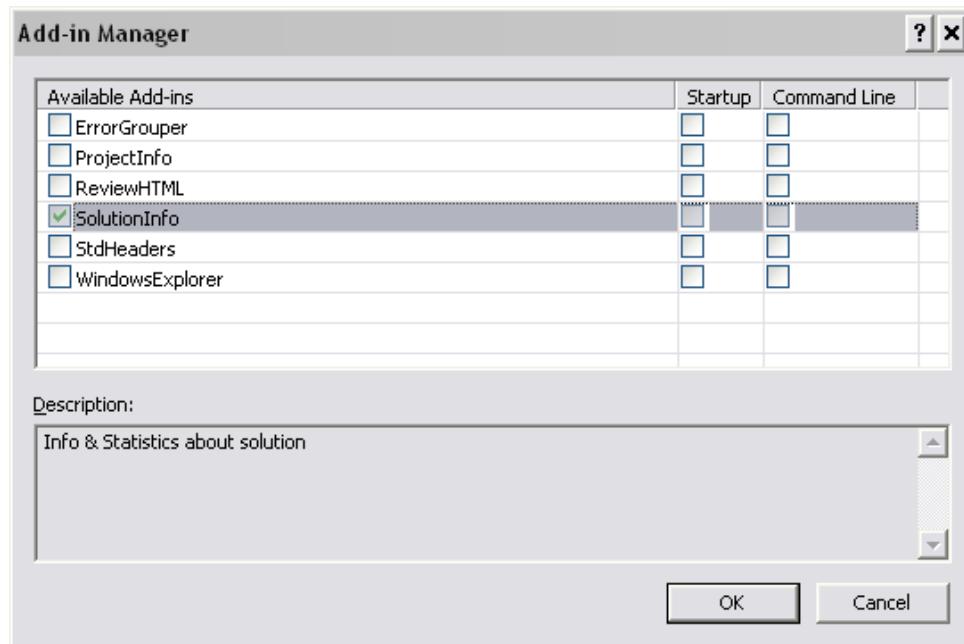


Figure 18: Add-in Manager

Whether the add-in module is enabled, whether it loads at start-up, and whether it can run from the command line is stored in the XML file. You can manipulate the XML file using the **Add-in Manager** window shown in the previous figure.



Tip: Clearing the add-in name does not immediately unload it from memory. You will most likely need to exit and restart Visual Studio to remove the DLL from memory.

If you install an add-in that does not behave and causes problems, you can start the IDE with the **/SafeMode** switch, which loads Visual Studio without any add-in modules at all.

Summary

Add-in module installation in Visual Studio versions 2008 and newer is very simple using the XML configuration option. You can make an install program or script if need be, but with an audience of primarily programmers, you can probably simply ask them to copy the two files to the appropriate folder.

Chapter 17 Object Reference

This chapter contains a summary of some of the basic object classes you can use to interact with Visual Studio.

Application Object (DTE2)

The application object (typically stored in the variable `_applicationObject`) is an encapsulation of everything within the Visual Studio IDE.

Property	Data Type	Description
ActiveDocument	Document	Currently active document.
ActiveSolutionProjects	Collection of projects	Collection of all projects in current solution.
ActiveWindow	Window	Currently active or topmost window.
Addins	Collection	Collection of all available add-ins.
CommandBars	Command Bar	Access to Visual Studio commands and menus.
CommandLineArgument	String	Command-line arguments passed to Visual Studio when it was started.
Debugger	Debugger	Access to Visual Studio debugger object.
DisplayMode	Enum	DisplayMDI or DisplayMDITabs.
Documents	Collection	Collection of open documents in the IDE.
Edition	String	Ultimate, Premium, Professional, or Express.
FullName	String	Full path and file name.
ItemOperations	Object	Allows file manipulation within Visual Studio.
LocaleID	Integer	Geographic region, 1033-United States, etc.
MainWindow	Window	Main window of the development environment.
Mode	Enum	IDE Mode Design or IDE Mode Debug.
RegistryRoot	String	Root key in registry where settings are stored.
Solution	Solution	Current solution object.
ToolWindows	Tool Window	Shortcut access object to IDE tool windows.

Property	Data Type	Description
Version	String	10.0, 12.0, etc.
Windows	Collection	Collection of all open IDE windows.

Windows and documents

Windows represent tool windows or editing forms used by Visual Studio. Tool windows include the **Solution Explorer**, **Properties**, the **Tool Box**, etc. Document windows are editing windows containing document objects that represent the source code being edited by the user. See [Chapters 9](#) and [10](#) for examples and more details.

Document

Property	Data Type	Description
ActiveWindow	Window	Window the document is open in.
FullName	String	Full path and file name of file in the document.
Kind	String	GUID string indicating type of document.
Name	String	Name of the document.
Path	String	Full path, without the file name.
ProjectItem	ProjectItem	Item within the project associated with the document.
Saved	Boolean	True if the document has not been modified since last open.
Selection	Selection	Current selection text in document.
Methods		
Activate	Move focus to the document.	
Close	Close, and optionally save the document.	
NewWindow	Create a new window to view the document.	
Object	Run-time object associated with the document.	
Redo	Re-execute last action that was undone.	
Save	Save the document to disk.	
Undo	Reverse last action performed on document.	

Window

The window represents either a tool window or a document window that contains text being edited.

Property	Data Type	Description
AutoHides	Boolean	Can the tool window be hidden?
Caption	String	The window title.
Document	Document	The document in the window (if one exists).
Height	Integer	Height of the window in pixels.
Kind	String	Either “Tool” or “Document”.
Left	Integer	Distance from left edge of the container in pixels.
Object	Object	Allow run-time access to contents in the window, most of Object(“TextDocument”).
ObjectKind	String	A GUID representing the tool contained in the window.
Project	Project	The project associated with the window.
ProjectItem	ProjectItem	The project element associated with the window.
Selection	Selection object	The currently selected text in the window.
Top	Integer	Distance from the top edge of the container in pixels.
Visible	Boolean	Is the window currently visible?
Width	Integer	Width of window, in pixels.
WindowState	Enum	vsWindowStateNormal, StateMinimize, or StateMaximize.
Methods		
Activate	Move focus to the window.	
Close	Close and optionally save the document.	
SetTabPicture	Set a picture to display for a tool window.	

Solution and projects

The solution object represents a solution and its component projects, which you can manipulate easily. See [Chapters 8](#) and [9](#) for more information.

Solution

The following table lists the solution properties of `_applicationObject`.

Property	Data Type	Description
AddIns	Collection	Collection of add-in objects associated with the solution.
FullName	String	Path and file name of the solution.
Globals	Collection	Global variables saved with solution.
IsOpen	Boolean	Is the solution open?
Projects	Collection	Collection of all projects in the solution.
Properties	Collection	Names and values of all solution properties.
Saved	Boolean	Has the solution been saved?
SolutionBuild	SolutionBuild	An object with build information of the solution.
TemplatePath	String object	Template path for type of project, i.e. C#, VB.
Methods		
AddFromFile	Add an existing file to the project.	
AddFromTemplate	Copy a template file and add it to the solution.	
Close	Close the solution.	
Create	Create an empty solution.	
FindProjectItem	Find an item in a project.	
Item	Get a project in the solution.	
Open	Open the solution.	
ProjectItemsTemplatePath	Return location of project item templates for specific project types.	
Remove	Remove specified project from solution.	
SaveAs	Save solution under another name.	

Project

You can iterate through the solution's `Projects` collection to get details on any given project in the solution space.

Property	Data Type	Description
CodeModel	Object	Code Model (access to source elements).
FullName	String	Full path and file name of the project.
Globals	Collection	Global add-in values associated with the project.
Kind	GUID String	Type of solution, VB or C#, for example.
Name	String	Short project name.
ProjectItems	Collection	Collection of items making up the project.
Properties	Collection	Properties associated with the project.
Saved	Boolean	Has the project been saved?
UniqueName	String	Unique name for the project.
Methods		
Save	Save the project or project item.	
SaveAs	Save as a new file name.	

Project Item

Project items are the files (source, XML files, etc.) that make up the project.

Property	Data Type	Description
ContainingProject	Project	The project hosting the project item or file.
Document	Object	A document object (if any exist) for the file.
FileCodeModel	Code Model	Allows you to access high-level code elements within the source file.
FileCount	Integer	Number of files associated with the project item.
FileNames	Collection	File names associated with the item.
IsOpen	Boolean	Is the project item open?
Kind	GUID	Type of the item.
Name	String	Name of the project item.
Object	Object	Run-time object associated with the project item.
Properties	Collection	Properties associated with the item.
Saved	Boolean	Has project item been modified since last open?

Methods	
Delete	Remove project from solution and storage.
ExpandView	Expand solution explorer view to show project.
Open	Open the project item.
Remove	Remove item from project and delete from disk.
Save	Save project item.
SaveAs	Save project item under another file name.

Code manipulation

These objects are the basic tools to manipulate text in source windows, both using simple string manipulations and the more complex code model parser. See [Chapters 12](#) and [13](#) for some example usage.

Text Document

Property	Data Type	Description
EndPoint	Text point	A point referring to the end of the document.
Selection	Object	The currently selected text.
StartPoint	Text point	A point referring to the start of the document.
Methods		
ClearBookmarks	Remove all unnamed bookmarks from the text document.	
CreateEditPoint	Create a point object to edit text within the document.	
MarkText	Create unnamed bookmarks for all found text in document.	
ReplacePattern	Replace patterns within entire document or range.	
ReplaceText	Simple text replacement with document.	

Edit Point

Property	Data Type	Description
AtEndOfDocument	Boolean	Is the point positioned at the document's end?

Property	Data Type	Description
AtEndOfLine	Boolean	Is the object at the end of the line?
AtStartOfDocument	Boolean	Is the point positioned at the start of the document?
AtStartOfLine	Boolean	Is the object at the start of a line?
CodeElement	Object	Return the code element at the current position.
DisplayColumn	Integer	The current column containing the point.
Line	Integer	The current line number in the document.
LineLength	Integer	Number of characters in the current line.
Methods		
ChangeCase	Change case of selected text.	
CharLeft	Move edit point specified number of characters to the left.	
CharRight	Move edit point specified number of characters to the right.	
ClearBookmark	Clear any unnamed bookmark on the current line.	
Copy	Copy range of text to clipboard.	
Cut	Copy text to clipboard and delete from document.	
Delete	Delete text from document.	
GetLines	Get lines of text between two lines.	
GetText	Get string of text.	
Insert	Insert text into document.	
LineDown	Move down one line.	
LineUp	Move up one line.	
MoveToLineAndOffset	Move to a line and character offset.	
Paste	Paste contents of clipboard at current point.	
ReplaceText	Replace selected text with given text.	
WordLeft	Move specified number of words to the left.	
WordRight	Move specified number of words to the right.	

Code Model

Using the code model is discussed in [Chapter 13](#).

Property	Data Type	Description
CodeElements	Collection	All the code element objects at this level.
IsCaseSensitive	Boolean	Is the current language case sensitive?
Language	String	Language the file is coded in.
Methods		
AddClass	Create a code class construct.	
AddEnum	Create an enum construct in the code.	
AddFunction	Create new function code.	
AddInterface	Create new interface code.	
AddNamespace	Create a new namespace in the module.	
AddVariable	Create new variable code.	
IsValidID	Is the specified identifier valid in the current language?	
Remove	Remove code element from file.	

Code Element

Property	Data Type	Description
Children	Collection	Collection of child code elements.
EndPoint	Text Point	Ending location for this element in file.
FullName	String	Fully qualified code element name.
InfoLocation	Enum	Is code element in project or external.
IsCodeType	Boolean	Can a code type object be obtained from the element?
Kind	Enum	Type of code element, i.e. class, function, etc.
Language	String	Language the code element is written in.
Name	String	Short name of the code element.
ProjectItem	ProjectItem	The project item associated with the code element.
StartPoint	Text Point	Starting location of the code element within the file.

Chapter 18 Add-in Helper Class

As you begin to work with add-ins, you might find yourself writing your own library of helper routines. Here is a sample class library to get you started.

We begin by declaring a variable in the helper class to hold a reference to the DTE2 object (**_applicationObject**), so we do not have to pass it around as a parameter.

```
using System;
using EnvDTE80;
using Microsoft.VisualStudio.CommandBars;
using System.Windows.Forms;

public class AddInsHelper
{
    public DTE2 app { get; set; }
```

MakeEmptySolution

While the solution object allows you to create an empty solution, it has a couple of changes to be aware of. The directory will not be made if it does not exist, and you need to save the new solution file. To isolate these behaviors, we can create our own method call to make an empty solution.

```
public void MakeEmptySolution(string folder, string SolName)
{
    string FullFolder;
    // Close solution if open.
    if (app.Solution.IsOpen)
        { app.Solution.Close(true); }
    // Get or make the folder for the solution.
    FullFolder = System.IO.Path.Combine(GetVSProjectsFolder(), folder);
    if (!System.IO.Directory.Exists(FullFolder))
    {
        System.IO.Directory.CreateDirectory(FullFolder); }
    string tempFile = System.IO.Path.Combine(FullFolder, "TempSolution.sln");
    app.Solution.Create(FullFolder, tempFile);
    tempFile = System.IO.Path.Combine(FullFolder, SolName);
    app.Solution.SaveAs(tempFile);
}
```

GetVSProjectsFolder

Another useful function is a wrapper to the **get_properties** method to return a path where Visual Studio stores new projects.

```

public string GetVSProjectsFolder()
{
    EnvDTE.Properties theProp = app.get_Properties("Environment", "ProjectsAndSolution");
    return theProp.Item("ProjectsLocation").Value.ToString();
}

```

FindMenuIndex

The **FindMenuIndex** method finds the index of a particular menu item on one of the main menu's menu bars. This allows you to control where to place your add-in module if you don't want it as the first or last item on the menu.

```

public int FindMenuIndex(string MainMenu, string subMenu)
{
    int res = 1;
    try
    {
        CommandBar menuBar = ((CommandBars)app.CommandBars)["MenuBar"];
        foreach (CommandBarController cb in menuBar.Controls)
        {
            if (MainMenu.ToString().ToUpper() ==
                cb.Caption.ToString().ToUpper().Replace("&", ""))
            {
                CommandBarPopup toolsPopup = (CommandBarPopup)cb;
                for (int xx = 1; xx <= toolsPopup.Controls.Count; xx++)
                {
                    if (toolsPopup.Controls[xx].Caption.ToString().ToUpper().Replace("&", "")
                        == subMenu.ToUpper())
                    {
                        res = toolsPopup.Controls[xx].Index;
                        break;
                    }
                }
                break;
            }
        }
    }
    catch
    {
    }
    return res;
}

```

Hopefully a few of these functions will help you get started writing your own tools and help support your add-in development projects.

Chapter 19 Third-Party Add-Ins

Ever since Microsoft provided outside parties the ability to create add-ins, thousands of third-party add-ins have been written and shared among developers. Many of the add-in modules are available for free or little cost. A good starting point is the Microsoft Visual Studio Gallery at <http://visualstudiogallery.msdn.microsoft.com/>.

Microsoft add-ins

Microsoft developers have provided a number of add-ins to the gallery. A few sample add-in programs available include:

- [Color Printing](#): Allows code files to be printed in color.
- [Regex Editor](#): IntelliSense, syntax color, and testing regular expressions. If you work with regular expressions, this add-in is a great addition to the IDE.
- [PowerCommands](#): Useful extensions to the Visual Studio IDE. Adds commands such as *Undo close*, *Insert GUID*, *Extract Constant*, etc. to the IDE.
- [Productivity Power Tools](#): Developer productivity extensions.
- Enhancements to the IDE, such as organize Visual Basic imports (similar to organize Usings in C#), align assignment statements, customize document tabs, etc.

It is not unusual to see the functionality of add-in modules developed internally by Microsoft make it into future releases of Visual Studio.

Community add-ins

There is a large community of programmers who are writing and sharing their add-ins to the website. Some useful add-ins from the community includes:

Indent Guides

The [Indent Guides](#) add-in displays vertical lines to show indentation levels. It provides a useful visual guide for aligning statements.

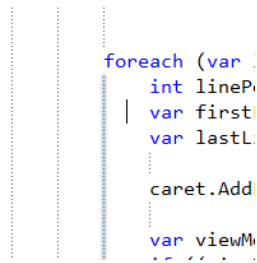


Figure 19: Indent Guide Add-in

Routing Assistant

The [Routing Assistant](#) add-in enables users to browse, define, match, and filter ASP.NET MVC routes for ASP.NET applications and websites with ease, directly from within Visual Studio. With the popularity of the MVC framework, this add-in module is a great time-saver and way to understand MVC routing behavior.

devColor

The [devColor](#) add-in underlines the colors in style sheets and includes a color picker dialog.

```
body
{
    background-color: #C00000;
}

a
{
    color: Lime;
}

table
{
    border: solid 1px RGB( 171 , 205 , 239);
}
```

Figure 20: devColor Add-in

The Visual Studio Gallery is a worthwhile site to visit and bookmark. You can also contribute your own add-ins if you create one that could be useful to other programmers.

Fame and glory await you!