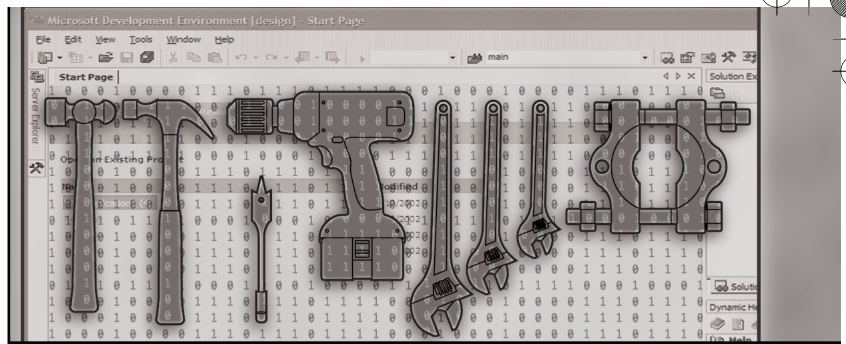
C04618747.fm Page 97 Friday, January 10, 2003 2:58 PM



4

Visual Studio .NET Macros

The macros facility in Microsoft Visual Studio .NET is arguably one of the most compelling reasons for using the IDE. This facility exposes almost all the functionality that you can access through the automation object model, but in an easy-to-use, scriptable form.

In this chapter, we'll introduce you to macros in Visual Studio .NET. We'll show you how to record macros and how to edit macro projects in the Macros IDE. We'll also show you how you can extend macros using .NET assemblies and how to share your macros with others. In addition, we'll explain how you can turn a macro project into a full-fledged Visual Studio .NET add-in, using a macro that ships with the Visual Studio .NET samples.

Macros: The Duct Tape of Visual Studio .NET

The macros facility of Visual Studio .NET uses Visual Basic .NET as its macro language. This fit has a much better feel to it than the Visual Basic Scripting Edition (VBScript) facility built into Microsoft Visual C++ 6.0. The Visual Basic .NET language can take full advantage of the .NET Framework and its own automation object model, so it offers an extremely powerful and compelling set of features that you can use to automate tasks in the IDE. In fact, you can convert any macro into a Visual Basic .NET–based add-in that you can compile and share with other developers.

As we mentioned in Chapter 1, Visual Studio .NET macros are saved into files with a .vsmacros extension. These macros are stored in the VSMacros71 folder in your default Visual Studio .NET projects folder. You can specify the Visual Studio .NET projects folder in the Options dialog box, on the Projects

C04618747.fm Page 98 Friday, January 10, 2003 2:58 PM

and Solutions page in the Environment folder. By default, this path is My Documents\Visual Studio Projects. Macros are stored in the VSMacros71 subfolder.

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

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

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

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

Recording Visual Studio .NET Macros

To record a Visual Studio .NET macro, first press the Ctrl+Shift+R keyboard shortcut. This combination brings up the Recorder toolbar and creates a macros module named *RecordingModule* if one doesn't already exist. You can see the

C04618747.fm Page 99 Friday, January 10, 2003 2:58 PM

Recorder toolbar in Figure 4-1. Notice that you can pause, stop, or even cancel the recording session that you've started.

Figure 4-1 The Recorder toolbar

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

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

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

We now have a *TemporaryMacro* method saved in the module *Recording-Module*. You can see that macro in Figure 4-2.

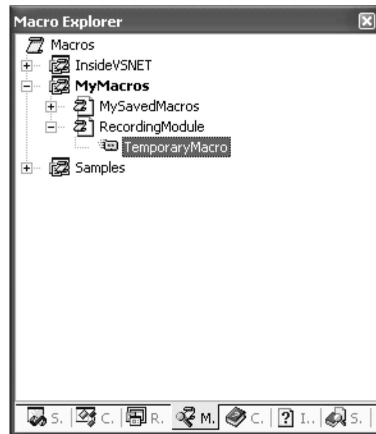


Figure 4-2 The Macro Explorer window

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

```
Imports EnvDTE
Imports System.Diagnostics
Public Module RecordingModule
    Sub TemporaryMacro()
        DTE.ExecuteCommand("Edit.Find")
        DTE.Find.FindWhat = "Connect"
        DTE.Windows.Item("Connect.cpp").Activate()
        DTE.Find.FindWhat = "Connect"
        DTE.Find.Target = vsFindTarget.vsFindTargetCurrentDocument
        DTE.Find.MatchCase = False
        DTE.Find.MatchWholeWord = False
        DTE.Find.Backwards = False
        DTE.Find.MatchInHiddenText = False
        DTE.Find.PatternSyntax = _
            vsFindPatternSyntax.vsFindPatternSyntaxLiteral
        DTE.Find.Action = vsFindAction.vsFindActionFind
        DTE.Find.Execute()
    End Sub
End Module
```

To play back this macro, press Ctrl+Shift+P, which is simply a shortcut to the *Macros.Macros.RecordingModule.TemporaryMacro* command. You should see the Find dialog box open with the first instance of the word you're searching for selected. In our case, this is the first instance of *Connect* in a file named Connect.cpp.

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

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

Macro Commands

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

Macros are divided into projects containing modules, which in turn contain methods. Projects are represented hierarchically in Macro Explorer below the Macro icon. Right-clicking the Macro icon brings up the shortcut menu containing commands for creating and loading macro projects. You can access the same functionality as named commands in the Command Window. Table 4-1 lists the macro commands related to macro projects.

Table 4-1 Macro Project Commands

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

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

Table 4-2 lists a few of the commands available from within a particular macro project.

Table 4-2 Macro Project Commands

Command	Description
Tools.Newmodule	Brings up the New Module dialog box, where you can create a new module from within Macro Explorer
Tools.Newmacro	When enabled, this command brings up the Macros IDE with a new macro method
Tools.Edit	Brings up the Macros IDE open to the currently selected project or module

By right-clicking on a macro in Macro Explorer, you can bring up a shortcut menu that lets you work with the macro directly. The Run command executes

the *Tools.Run* command on the currently selected macro. The Rename command allows you to edit the name of the macro in place. The change you make to the name is reflected in the method name in the Macros IDE. The Delete command deletes the currently selected macro. And finally, the Edit command opens the current macro in the Macros IDE.

Macro Explorer is a powerful tool for organizing the macros you've created. You'll find that you can do quite a bit in Macro Explorer without having to go to the Macros IDE. For example, you can record a macro, rename that macro to save it, and even add that same macro to a toolbar or a menu in the IDE, all without having to go to the Macros IDE. You'd probably find it limiting not to use the IDE, but it is possible. To really get the most out of Visual Studio .NET macros, you'll want to be able to create and edit them from within the Macros IDE.

Editing Macros in the Macros IDE

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

One difference you'll notice right away is that all your loaded macro projects show up in the Project Explorer window. Visual Studio .NET ships with an extremely useful set of macros out of the box. You can see these macros if you expand the Samples project in Project Explorer in the Macros IDE (as shown in Figure 4-3).

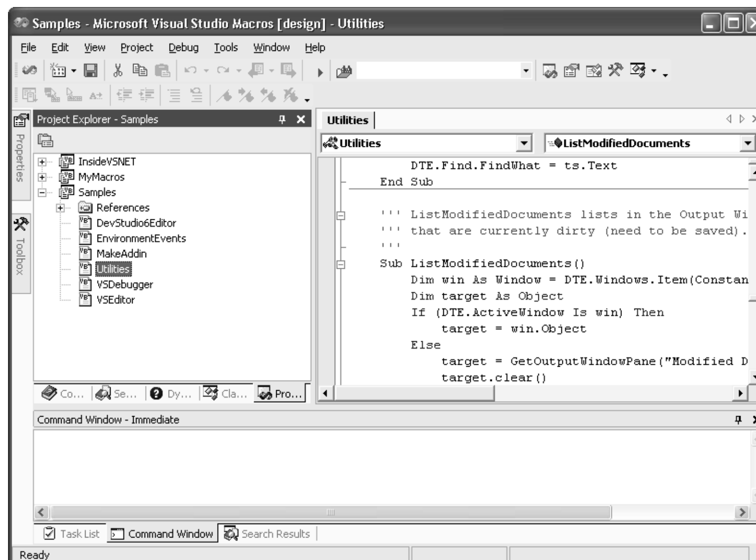


Figure 4-3 The Samples project in the Macros IDE

C04618747.fm Page 103 Friday, January 10, 2003 2:58 PM

The memory space for macro projects is separated, so if you want to utilize functionality between different macros or if you want to take advantage of a common set of environmental events, you must keep the macros that you write inside the same project. If you want to access functionality from another macro project, you simply copy the macros you want to access into the project you're working on. For example, you can copy modules from the Samples project into your own project to take advantage of the functionality exposed by those macros.

To create a new macro project, you need to start from the Visual Studio .NET IDE. You can use the New Macro Project command on the shortcut menu in Macro Explorer or you can enter **Tools.NewMacroProject** into the Command Window to open the New Macro Project dialog box (shown in Figure 4-4). Enter a name and location for your project, and then click OK. Pressing Alt+F11 will toggle you back to the Macros IDE, where you can work on the code in the new project.

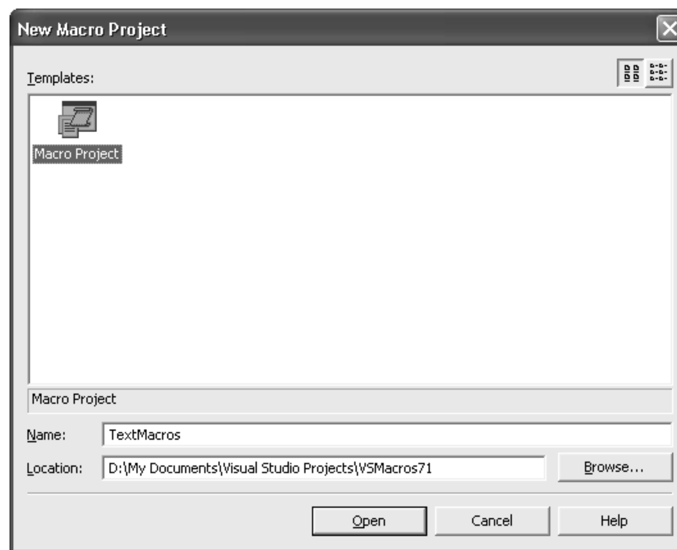


Figure 4-4 The New Macro Project dialog box

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

C04618747.fm Page 104 Friday, January 10, 2003 2:58 PM

Lab: Navigating Between IDEs

To shift from the Macros IDE to the Visual Studio .NET IDE, you can click the Visual Studio button on the Macros IDE toolbar. There's no such button on any of the default Visual Studio .NET toolbars, so you'll need to add one if you want to get back to the Macros IDE in the same way. To do so, right-click on a toolbar in the Visual Studio .NET IDE and click Customize. On the Commands tab, find the Macros IDE command in the Tools category and drag it to the toolbar you want to use it from. The button will have the same infinity image used in the Macros IDE. This makes it easy to navigate between the two IDEs while you work on your macros.

If you'll be doing a lot of macro development, a better solution is to run your machine with two monitors, keeping the Macros IDE in one screen and Visual Studio .NET in the other.

Adding a reference to a macro project is slightly different from adding one to a standard Visual Basic .NET project. If you look at the Add Reference dialog box that's used in the Macros IDE Project Explorer (shown in Figure 4-5), you'll notice that it doesn't offer a way to add custom assemblies.

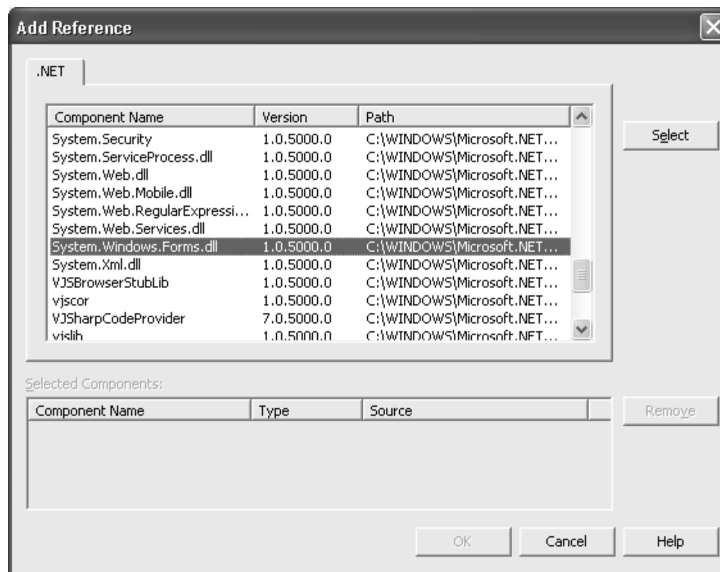


Figure 4-5 Add Reference dialog box

C04618747.fm Page 105 Friday, January 10, 2003 2:58 PM

To add references to your own assemblies, you must copy them to the C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\PublicAssemblies folder. You can then add your own reference to the assembly from the Add Reference dialog box. Using assemblies, you can write your macro functionality in any language you want and then access that functionality from a fairly simple macro. You can also write assemblies that call to unmanaged code and assemblies that act as COM wrappers to access COM functionality from within your macros.

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

A Simple Macro

Earlier in the book, we touched on the behavior of the *File.NewFile* command in Visual Studio .NET. Some programmers haven't been pleased that this command displays a dialog box by default, forcing them to resort to the mouse or to a series of keystrokes to get an empty file up and running. But the solution is simple: you just create a macro that does exactly what you want and then assign that macro an alias in the Command Window. The following code is all you really need to create a new text file in the IDE:

```
Imports EnvDTE
Imports System.Diagnostics

Public Module NewFile

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

End Module
```

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

To get to a macro you want to execute, you've got a few choices. One approach is to run the macro from Macro Explorer in the Visual Studio .NET IDE. This works fine, but it's probably not the optimal solution for a macro that you're planning to use often. The second choice is to create an alias for the

C04618747.fm Page 106 Friday, January 10, 2003 2:58 PM

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

```
>alias nf Macros.InsideVSNET.Chapter04.NewFile.NewTextFile
```

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

The *Imports* statement in this sample is important. The API associated with the Visual Studio .NET automation object model is contained in the *EnvDTE* namespace. The automation object model is discussed in depth in Chapter 5 through Chapter 12. Here we simply want to familiarize you with this object model and get you up and running with some of the more common functionality that you'll use in your macro projects. Most of the subjects covered in the chapters that comprise Part II of the book apply to both macros and add-ins. In fact, you can use macros to quickly test add-in functionality that you're writing. You'll save time because you normally test an add-in by compiling the add-in and loading a second instance of the IDE. Using a macro, you can get to the automation object model, write and test your routines, and then add them to your add-in projects.

Working with Macros

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

C04618747.fm Page 107 Friday, January 10, 2003 2:58 PM

Manipulating Documents and Text

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

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

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

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

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

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

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

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

End Sub
```

C04618747.fm Page 108 Friday, January 10, 2003 2:58 PM

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

Let's take a look at a second, more useful, example that inserts text into a document based on a selection. The following example creates an HTML comment in a document. This functionality doesn't exist in Visual Studio .NET 2003, so you might find this simple macro useful enough to add to your own toolbox. Here we'll declare *ts* as a *TextSelection* object and assign it the current selection using *DTE.ActiveDocument.Selection*:

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

This macro uses the *TextSelection Insert* method to insert text around the *Selection* object. The *Insert* method takes two arguments. The first argument is the string that you want to insert into the selection. The second argument is a *vsInsertFlags* constant that defines where the insertion is to take place. The first *Insert* call in the example uses *vsInsertFlagsAtStart*. The second uses *vsInsertFlagsAtEnd*. Table 4-3 lists these constants.

Table 4-3 *vsInsertFlags* Constants

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

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

C04618747.fm Page 109 Friday, January 10, 2003 2:58 PM

Moving Windows

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

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

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

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

C04618747.fm Page 110 Friday, January 10, 2003 2:58 PM


```
        ' If our output pane exits, we'll use that to output the string,
        ' otherwise, we'll add it to the list.
        If exists Then
            owp = ow.OutputWindowPanes().Item(i)
        Else
            owp = ow.OutputWindowPanes.Add("MsgWin Output")
        End If
        ' Here we set the Output window to visible, activate the pane,
        ' and send the string to the pane.
        win.Visible = True
        owp.Activate()
        owp.OutputString(msg + vbCrLf)
    End If
End Sub
```

MsgWin uses a pretty cool feature that's found in the samples that ship with Visual Studio .NET. The method determines whether the calling method was invoked from the Command Window. If it was, the output is directed right back to the user in the Command Window. If it's called from a macro that was run from a menu, shortcut, or button, *MsgWin* sends the output to an Output window named MsgWin Output.

Tip The Samples macros project that ships with Visual Studio .NET contains a lot of really good, functional macro code that you can use in the macros you write.

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

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

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

Figure 4-6 shows what the Visual Studio .NET IDE looks like after it has been invoked from the *MsgWinText* macro in the IDE.

C04618747.fm Page 111 Friday, January 10, 2003 2:58 PM

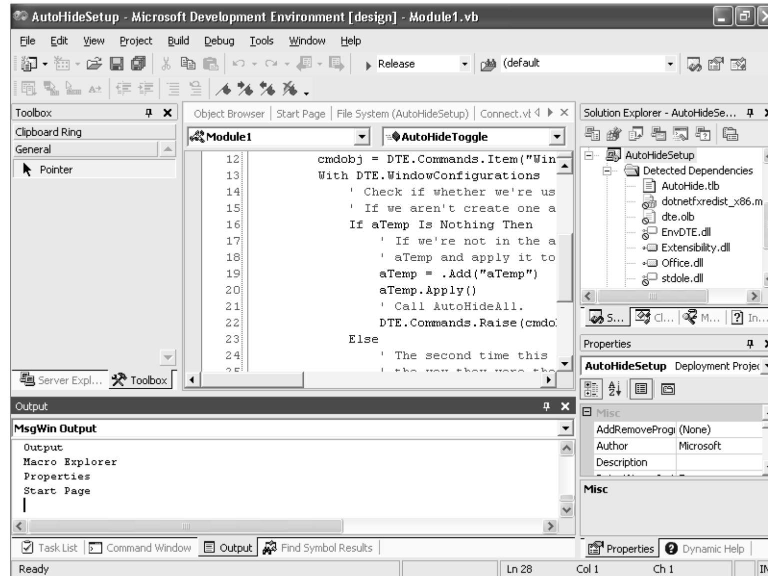


Figure 4-6 The MsgBox Output window in the IDE

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

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

Of course, this overload won't do you much good if you call the macro the way we did in *MsgBoxTest*, but as you can see it's easy enough to do what you want with the macro.

Another way to add this kind of functionality to your macros is to create an assembly in the language of your choice and then reference that assembly from within your macro project. We did this with the *CommandWindowPaneEx* object.

Using Assemblies in Your Macros

A couple of things become apparent when you start to use macros a lot. The first is that you can use macros as a place to test the functionality of .NET assemblies. For example, if you want to test some bit of functionality in the framework, all you need to do is reference the appropriate assembly and then call the methods from within a macro. With a little practice, you'll find that the Macros IDE can work as a little laboratory that lets you try out functionality without having to mess around with rebuilding your projects.

The second thing you'll notice is that you have to write all this cool stuff in Visual Basic, and if that's not your preferred language, you might be spending a lot of time performing tasks you already know how to accomplish quickly in another language. As we mentioned earlier, there is a way to write macro functionality in languages other than Visual Basic—by building your functionality into an assembly and then referencing that assembly from within your macro project.

We wrote a base set of utility functions for the book that you can take advantage of in your own macros and add-ins. In the Utilities folder of the companion content, you'll find the Utilities solution. This solution contains the *OutputWindowPaneEx* object. Build the solution and copy the InsideVSNET.Utilities.dll file from the bin\debug folder for the project into the C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\PublicAssemblies folder.

Once you've copied that file into your Public Assemblies folder, you can add a reference to the assembly from your macro project in the Macros IDE by right-clicking on the References folder in the Project Explorer window and choosing Add Reference, or by selecting the References folder and typing **Project.AddReference** into the Macros IDE Command Window. On the .NET tab of the Add Reference dialog box, you should see the new InsideVSNET.Utilities assembly. Select it in the list, click Select, and then click OK. You'll see the new assembly in the list of references if you expand the References folder.

After you add the reference to your project, it's helpful to add the appropriate *Imports* statement to your module. In this case, you'll add the following to the top of the module:

```
Imports InsideVSNET.Utilities
```

C04618747.fm Page 113 Friday, January 10, 2003 2:58 PM

Once you add the reference, IntelliSense kicks in automatically as you create an *OutputWindowPaneEx* object and use it. The really cool thing about this object is that it lets you specify whether to send your output to the Output window in the Visual Studio .NET IDE or to the Macros IDE. In this example, we specified the Macros IDE, passing *DTE.MacrosIDE* when we created the object. We also changed the test a bit by enumerating the open windows in the Macros IDE rather than the Visual Studio .NET IDE, as we did earlier.

```
Sub OutputWindowPaneExTest()
    Dim owp As New OutputWindowPaneEx(DTE.MacrosIDE)
    Dim wins As Windows = DTE.MacrosIDE.Windows()
    Dim i As Integer

    owp.Activate()

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

In addition to letting you perform the tasks that you're able to perform with the *OutputWindowPane* object, *OutputWindowPaneEx* lets you do a number of things with text that you want to send to an Output window. As we mentioned, you can specify the IDE to which you want to send your output. The *Write* method has three overloads, letting you specify an object, a string, or a formatting string/parameter array. Using these overloads, you can specify formatting options, much like using the *System.Console.Write* and *System.Console.WriteLine* methods in the .NET Framework.

Macro Events

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

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

C04618747.fm Page 114 Friday, January 10, 2003 2:58 PM

a block of code that's been generated automatically by the IDE. Here's the important part of the block. (The attributes have been removed to make this fit the page.)

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

As you can see from this listing, there are a lot of event types you can take advantage of in the IDE. In fact, you can use all the DTE events, though they're not included by default. You can add these other events to this list to get to the events that you're interested in. To create a new event handler, you select the event type you want to handle from the Class Name list at the top of the code window. You can see how this looks in Figure 4-7.

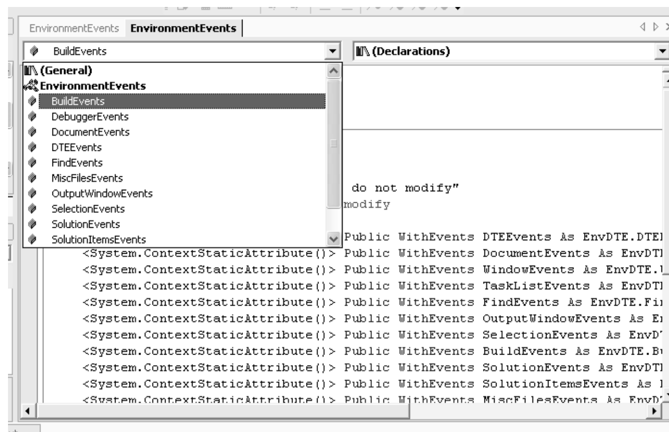


Figure 4-7 Selecting the event type you want to handle from the Class Name list

C04618747.fm Page 115 Friday, January 10, 2003 2:58 PM

After you select an event type, the Method Name list in the upper-right portion of the code pane will list the events you can handle, as shown in Figure 4-8.

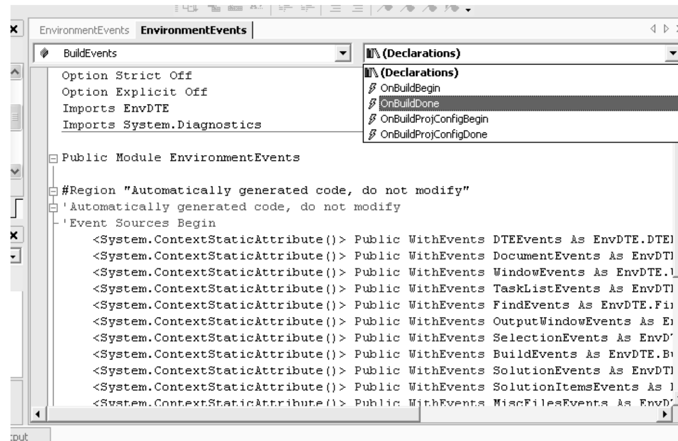


Figure 4-8 Selecting the event you want to handle from the Method Name list

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

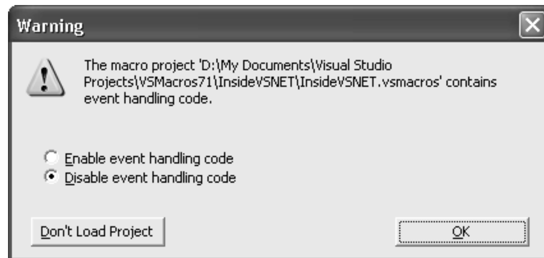
```
Private Sub BuildEvents_OnBuildDone(ByVal Scope As EnvDTE.vsBuildScope, _
    ByVal Action As EnvDTE.vsBuildAction) _
    Handles BuildEvents.OnBuildDone
        MsgWin("Build is done!")
End Sub
```

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

C04618747.fm Page 116 Friday, January 10, 2003 2:58 PM

Event Security

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



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

Sharing Macros with Others

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

Exporting Modules and Projects

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

To export a macro module in Visual Studio .NET, you must open the Macros IDE and select the module you want to export from the Project Explorer window. Pressing Ctrl+E will invoke the *File.SaveSelectedItemsAs* command,

C04618747.fm Page 117 Friday, January 10, 2003 2:58 PM

which brings up the Export File dialog box. This command is listed on the File menu as Export.

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

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

To make a macros project text-based, you change the *StorageFormat* property in the Visual Studio .NET IDE for the project that you want to change. Select the project in Macro Explorer and then change the *StorageFormat* property in the Properties window from Binary (.vsmacros) to Text (UNICODE). This change will create a number of files in the macro project's folder that looks much like a regular Visual Studio .NET project folder. In Figure 4-9, you can see the folder for the Samples project after it has been converted to Text format.

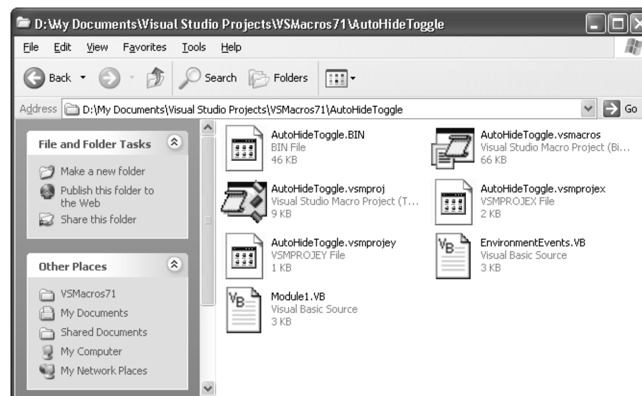


Figure 4-9 A macro project that has been stored in Text format

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

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

Also keep in mind that shipping a binary macro project does nothing to safeguard your source code. To do that, you're better off adding your functionality to

C04618747.fm Page 118 Friday, January 10, 2003 2:58 PM

an assembly that gets called from a macro, as we demonstrated earlier. An even better solution, where appropriate, is to turn your macro project into an add-in.

Turning Macros into Add-ins

Visual Studio .NET ships with a macro in the Samples project that lets you turn a macro project into a Visual Studio .NET add-in, complete with an installer. The macros in the *MakeAddin* module take a macro and turn it into an add-in project that you can compile and install into your environment. Macros that have been turned into add-ins can be shipped as binaries that are installed on a user's machine. Keep in mind, though, that you really should test the add-ins that you create in this way to make sure they do what you expect.

For this example, we'll take the *AutoHideToggle* macro that we created in Chapter 1 and turn it into an add-in using the *MakeAddin* macros. To get started, copy the macro you want to turn into an add-in into a new macro project. You can give the project the same name as the method if you want. Just be sure to build the new project to test it before you use it.

Next, you create a new add-in project in Visual Studio .NET by pressing Ctrl+Shift+N to open the New Project dialog box. Expand the Other Projects folder and then open the Extensibility Projects folder to get to the Visual Studio .NET Add-in project template, as shown in Figure 4-10.

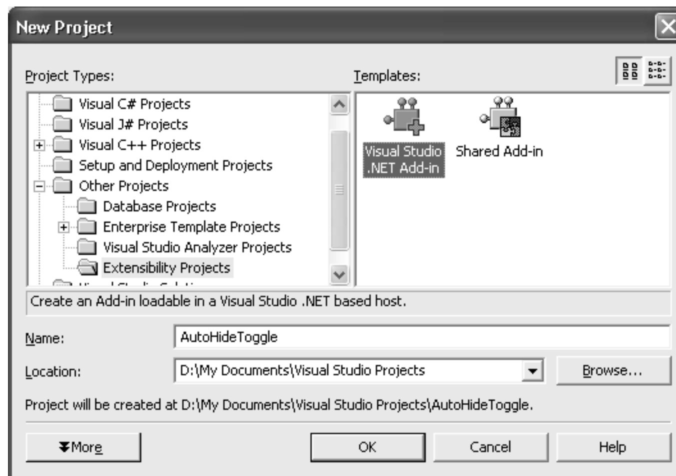


Figure 4-10 Creating a new add-in project

Give your new add-in an appropriate name, and then go through the wizard to create a Visual Basic .NET add-in project that installs a menu command in the IDE.

C04618747.fm Page 119 Friday, January 10, 2003 2:58 PM

After your new add-in project is complete, press Ctrl+Alt+A to bring up the Command Window and then type **Macros.Samples.MakeAddin.MakeAddin-FromMacroProj**. This will bring up the input box shown in Figure 4-11. In the box, type the name of the macro project you want to turn into an add-in, and then click OK.

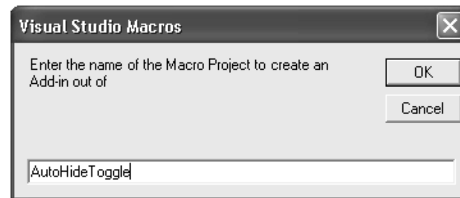


Figure 4-11 Specifying the macro project you want to turn into an add-in

At this point, if all the projects are of the right type, you should see a message asking you to confirm that you want to run the macro on the current project. Click OK, and the *MakeAddin* macro project will complete its work. When the macro is finished, you'll see the dialog box shown in Figure 4-12. What the *MakeAddin* macro has done is add the macro functionality from your project to the new add-in.

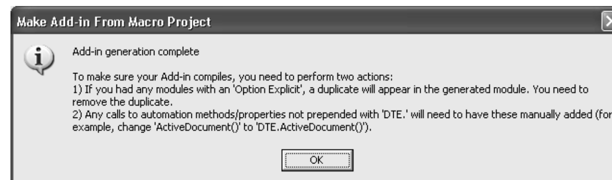


Figure 4-12 The message box confirming completion and final instructions for the *MakeAddin* macro

To test the new add-in, press F5 to start debugging; you should see a second instance of the Visual Studio .NET IDE open. On the Tools menu of that second instance of the IDE, you should see a menu containing the name of your project and the name of the macros that were in your original macro project. You can see in Figure 4-13 that the AutoHideToggle add-in is ready to go. Choosing this menu command does exactly what running the macro in the IDE did in Chapter 1.

C04618747.fm Page 120 Friday, January 10, 2003 2:58 PM

> **Note** If you use this add-in or the macro described in Chapter 1, keep in mind that the layout you have open will become your default layout. It won't overwrite the layouts that ship with Visual Studio .NET, but if you want to get back to one of those, you'll need to go back to the My Profile tab on the Start Page and select an alternative layout in the Windows Layout list. Then select the layout you chose when you originally created your profile and your windows should return to that layout.

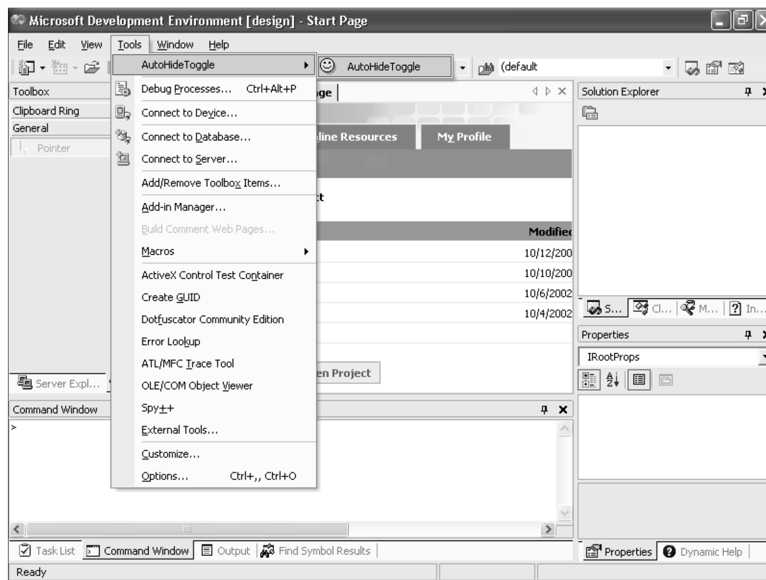


Figure 4-13 The new AutoHideToggle add-in in the IDE

To use this new add-in in Visual Studio .NET, you build a release version of the setup project that's generated for the add-in and then you navigate to the folder containing the setup program. Close all your instances of Visual Studio .NET and run the Setup.exe program. When you open the IDE, you'll find that your new add-in has been installed and is ready for use.

C04618747.fm Page 121 Friday, January 10, 2003 2:58 PM

Looking Ahead

This chapter gave you some information about using the Visual Studio .NET macro facility to perform some simple automation tasks. Part II of the book should provide you with enough information to do just about anything you want with automation in Visual Studio .NET.