Introduction to Office Solutions

The Three Basic Patterns of Office Solutions

Now that you understand the basic pattern of the Office object models, this chapter explains how developers pattern and build their Office solutions. Most solutions built using Office follow one of three patterns:

- Office automation executable
- Office add-in
- Code behind an Office document

An automation executable is a program separate from Office that controls and automates an Office application. An automation executable can be created with development tools such as Visual Studio. A typical example is a standalone console application or Windows Forms application that starts up an Office application and then automates it to perform some task. To start a solution built this way, the user of the solution starts the automation executable, which in turn starts up the Office application. Unlike the other two patterns, the automation code does not run in the Office process but runs in its own process and talks cross process to the Office process being automated.

An add-in is a class in an assembly (DLL) that Office loads and creates when needed. An add-in runs in process with the Office application instead of requiring its
own process separate from the Office application process. To start a solution built this way, the user of the solution starts the Office application associated with the add-in. Office detects registered add-ins on startup and loads them. An add-in can customize an Office application in the same ways that code behind a document can. However, code behind a document unloads when the document associated with the code is closed—an add-in can remain loaded throughout the lifetime of the Office application.

The code behind pattern was popularized by Visual Basic for Applications (VBA)—a simple development environment that is included with Office that enables the developer to write Visual Basic code against the object model of a particular Office application and associate that code with a particular document or template. A document can be associated with C# or Visual Basic code behind using Visual Studio 2005 Tools for Office (VSTO). To start a solution built this way, the user of the solution opens a document that has code behind it or creates a new document from a template that has code behind it. The code behind the document will customize the Office application in some way while the document is open. For example, code behind the document might add menu items that are only present when the document is open or associate code with events that occur while the document is open.

We discuss two advanced patterns later in this book. The server document pattern involves running code on a server to manipulate data stored in an Office document without starting the Office application. VSTO makes this scenario possible through a feature called cached data. Chapter 18, “Server Data Scenarios,” discusses this pattern. The XML and XSLT pattern is similar to the server document pattern and involves writing code to generate Word or Excel documents in WordprocessingML or SpreadsheetML format without starting the Office application. You can also generate these formats by applying an XSLT transform to some XML data. Chapters 21, “Working with XML in Excel,” and 22, “Working with XML in Word,” discuss these scenarios.

**Hosted Code**

The add-in and code behind patterns are sometimes called hosted code, which means that your code runs in the same process as the Office application.
**Discovery of Hosted Code**

For code to run in the Office application process, the Office application must be able to discover your code, load the code into its process space, and run your code. Office add-ins are registered in the Windows registry so that Office can find and start them. Using the registry seems a little non-.NET, but this is necessary because Office 2003 talks to add-ins as if they were COM objects through COM interop.

The code behind a document pattern does not require a registry entry. Instead, code is associated with a document by adding some special properties to the document file. Office reads these properties when the document opens, and then Office loads the code associated with the document.

**Context Provided to Hosted Code**

It is critical that your hosted code get context—it needs to get the Application object or Document object for the Office application into which it is loading. COM add-ins are provided with context through an interface implemented by the add-in class. Outlook add-ins in VSTO are provided with context through a class created in the project that represents the application being customized. Code behind a document in VSTO is provided with context through a class created in the project that represents the document being customized.

**Entry Point for Hosted Code**

At startup, Office calls into an entry point where your code can run for the first time and register for events that might occur later in the session. For a COM add-in, this entry point is the OnConnection method of the IDTExtensibility2 interface implemented by the COM add-in. For a VSTO Outlook add-in and VSTO code behind a document, this entry point is the Startup event handler.

**How Code Gets Run After Startup**

After hosted code starts up, code continues to run in one or more of the following ways.
**Code Runs in Response to Events Raised by Office**

The most common way that code runs after startup is in response to events that occur in the Office application. For example, Office raises events when a document opens or a cell in a spreadsheet changes. Listing 1-24 shows a simple class that listens to the Activate event that Excel’s Worksheet object raises. Typically, you will hook up event listeners, such as the one shown in Listing 1-24, when the initial entry point of your code is called.

**Interface Methods Called on Objects Provided to Office**

Objects such as the startup class for a COM add-in implement an interface called IDTExtensibility2 that has methods that Office calls during the run of the Office application. For example, if the user turns off the COM add-in, Office calls the OnDisconnection method on the IDTExtensibility2 interface implemented by the COM add-in. In this way, additional code runs after the initial entry point has run.

**Events Raised on Code Behind Classes**

The classes generated in VSTO projects that represent the customized application or document handle the Startup and Shutdown events. After the constructor of the class executes, Office raises the Startup event. When the document is about to be closed, Office raises the Shutdown event.

**How Code Gets Unloaded**

Your code gets unloaded in a number of ways, depending on the development pattern you are using. If you are using the automation executable pattern, your code unloads when the automation executable you have written exits. If you are using the add-in pattern, your code unloads when the Office application exits or when the user turns off the add-in via an add-in management dialog. If you are using the code behind pattern, your code unloads when the document associated with your code is closed.

In the hosted patterns of running code, there is some method that is called or event that is raised notifying you that you are about to be unloaded. For COM add-ins, Office calls the OnDisconnection method. For VSTO code behind documents and Outlook add-ins, Office raises the Shutdown event before your code is unloaded.
Office Automation Executables

This section considers each of these three patterns of Office solutions in more detail. Office solutions that use the automation executable pattern start up an Office application in a very straightforward manner—by creating a new instance of the Application object associated with the Office application. Because the automation executable controls the Office application, the automation executable runs code at startup and any time thereafter when executing control returns to the automation executable.

When an automation executable uses `new` to create an Application object, the automation executable controls the lifetime of the application by holding the created Application object in a variable. The Office application determines whether it can shut down by determining the reference count or number of clients that are using its Application object.

In Listing 2-1, as soon as `new` is used to create the `myExcelApp` variable, Excel starts and adds one to its count of clients that it knows are holding a reference to Excel’s Application object. When the `myExcelApp` variable goes out of scope (when Main exits), .NET garbage collection releases the object and Excel is notified that the console application no longer needs Excel’s Application object. This causes Excel’s count of clients holding a reference to Excel’s Application object to go to zero, and Excel exits because no clients are using Excel anymore.

When you create an Office application by creating a new instance of the Application object, the application starts up without showing its window, which proves useful because you can automate the application without distracting the user by popping up windows. If you need to show the application window, you can set the Visible property of the Application object to `true`. If you make the main window visible, the user controls the lifetime of the application. In Excel, the application will not exit until the user quits the application and your variable holding the Excel Application object is garbage collected. Word behaves differently—the application exits when the user quits the application even if a variable is still holding an instance of the Word Application object.

Listing 2-1 sets the status bar of Excel to say “Hello World” and opens a new blank workbook in Excel by calling the Add method of Excel’s Workbooks collection.

Listing 2-1 Automation of Excel via a Console Application

```csharp
using System;
using Excel = Microsoft.Office.Interop.Excel;
using System.Windows.Forms;

namespace ConsoleApplication
{
    class Program
    {
        static bool exit = false;

        static void Main(string[] args)
        {
            Excel.Application myExcelApp = new Excel.Application();
            myExcelApp.Visible = true;
            myExcelApp.StatusBar = "Hello World";
            myExcelApp.Workbooks.Add(System.Type.Missing);

            myExcelApp.SheetBeforeDoubleClick +=
                new Excel.AppEvents_SheetBeforeDoubleClickEventHandler(
                    myExcelApp_SheetBeforeDoubleClick);

            while (exit == false)
            {
            }

            static void myExcelApp_SheetBeforeDoubleClick(object sheet,
                Excel.Range target, ref bool cancel)
            {
                exit = true;
            }
        }
    }
}
```

Listing 2-1 also illustrates how an automation executable can yield time back to the Office application. A reference to the System.Windows.Forms assembly must be added to the project. After event handlers are hooked up, System.Windows.Forms.Application.DoEvents() is called in a loop to allow the Excel application to run normally. If the user double-clicks a cell, Office yields time back to the event handler in the automation executable. In the handler for the Double-Click event, we
set the static variable `exit` to `true`, which will cause the loop calling `DoEvents` to exit and the automation executable to exit.

You can see the lifetime management of Excel in action by running the automation executable in Listing 2-1 and exiting Excel without double-clicking a cell. Excel will continue to run in a hidden state, waiting for the console application to release its reference to Excel’s Application object.

**Creating a Console Application That Automates Word**

This section walks you through the creation of a simple console application that automates Word to create a table specified in wiki text format. A wiki is a kind of online encyclopedia that users can contribute to. For an example, see [http://www.officewiki.net](http://www.officewiki.net) for a wiki that documents the Office primary interop assemblies (PIAs). Wikis use simple, easy-to-edit text files that any visitor to the wiki can edit without having to know HTML. These text files have simple representations of even complex elements such as tables. Our console application will read a simple text file that specifies a table in wiki text format. It will then automate Word to create a Word table that matches the text file specification.

In the wiki text format, a table that looks like Table 2-1 is specified by the text in Listing 2-2.

**Table 2-1 A Simple Table Showing the Properties and Methods of Word’s Add-In Object**

<table>
<thead>
<tr>
<th>Property or Method</th>
<th>Name</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>Application</td>
<td>Application</td>
</tr>
<tr>
<td>Property</td>
<td>Autoload</td>
<td>Boolean</td>
</tr>
<tr>
<td>Property</td>
<td>Compiled</td>
<td>Boolean</td>
</tr>
<tr>
<td>Property</td>
<td>Creator</td>
<td>Int32</td>
</tr>
<tr>
<td>Method</td>
<td>Delete</td>
<td>Void</td>
</tr>
<tr>
<td>Property</td>
<td>Index</td>
<td>Int32</td>
</tr>
</tbody>
</table>
We will use Visual Studio 2005 to create a console application. After launching Visual Studio, choose New Project from the File menu. The New Project dialog box shows a variety of project types. Choose the Visual C# node from the list of project types, and choose the Windows node under the Visual C# node. This is slightly counterintuitive because there is an Office node available, too, but the Office node only shows VSTO code behind document projects and the VSTO Outlook add-in project.

After you choose the Windows node, you will see in the window to the right the available templates. Choose the Console Application template. Name your console application project, and then click the OK button to create your project. In Figure 2-1, we have created a console application called WordWiki. Note that the new project dialog can have a different appearance than the one shown in Figure 2-1 depending on the profile you are using. In this book, we assume you are using the Visual C# Development Settings profile. You can change your profile by choosing Import and Export Settings from the Tools menu.
When you click the OK button, Visual Studio creates a console application project for you. Visual Studio displays the contents of the project in the Solution Explorer window, as shown in Figure 2-2.

Figure 2-1 Creating a console application from the New Project dialog.

When you click the OK button, Visual Studio creates a console application project for you. Visual Studio displays the contents of the project in the Solution Explorer window, as shown in Figure 2-2.

Figure 2-2 The Console application project WordWiki shown in Solution Explorer.
By default, a newly created console application references the assemblies System, System.Data, and System.Xml. We also need to add a reference to the Word 2003 PIA. We do this by right-clicking the References folder and choosing Add Reference from the pop-up menu that appears. This shows the Add Reference dialog in Figure 2-3. Click the COM tab and choose the Microsoft Word 11.0 Object Library to add a reference to the Word 2003 PIA, and then click the OK button.

![Figure 2-3 Adding a reference to the Microsoft Word 2003 PIA.](image)

Visual Studio adds the reference to the Word 2003 PIA and adds additional references to the stdole, VBIDE, and Microsoft.Office.Core PIAs, as shown in Figure 2-4. These additional PIAs are ones that the Word PIA depends on. Stdole is a PIA that contains the definition of some of the types that COM object models need. VBIDE is the PIA for the object model associated with the VBA editor integrated into Office. Microsoft.Office.Core (office.dll) is the PIA for common functionality shared by all the Office applications, such as the object model for the toolbars and menus.

Now that the proper references have been added to the console application, let’s start writing code. Double-click Program.cs in the Solution Explorer window to edit the main source code file for the console application. If you have outlining turned on, you will see the text “using …” at the top of the Program.cs file with a + sign next to it. Click the + sign to expand out the code where the using directives are placed.
Add the following three using directives so that you can more easily use objects from the Word PIA and the Microsoft.Office.Core PIA as well as classes in the System.IO namespace.

```csharp
using Office = Microsoft.Office.Core;
using Word = Microsoft.Office.Interop.Word;
using System.IO;
```

We alias some of these namespaces so we do not have to type out the entire namespace, such as Microsoft.Office.Interop.Word, every time we want to declare a Word object. With the alias in place, we can just type Word to specify the namespace. We keep an alias namespace in place for Word and Office instead of just typing `using Microsoft.Office.Interop.Word` and importing all the types into global scope. This is because Word and Office define hundreds of types, and we do not want all these type names potentially colliding with types we define in our code or with other referenced types. Also for the purpose of this book, the code is clearer when it says `Word.Application` rather than `Application`, so you know what namespace the `Application` type is coming from.

We are now ready to write some code that automates Word to create a table after reading a text input file in the wiki table format. Listing 2-3 shows the entire listing of our program. Rather than explain every line of code in that listing, we focus on the lines of code that automate Word. We assume the reader has some knowledge of how to read a text file in .NET and parse a string via the Split method. We briefly touch on
some objects in the Word object model here, but Chapters 6 through 8—“Programming Word,” “Working with Word Events,” and “Working with Word Objects,” respectively—cover the Word object model in much more detail.

The first thing we do in Listing 2-3 is declare a new instance of the Word application object by adding this line of code to the Main method of our program class.

```csharp
Word.Application theApplication = new Word.Application();
```

Although Word.Application is an interface, we are allowed to create a new instance of this interface because the compiler knows that the Word.Application interface is associated with a COM object that it knows how to start. When Word starts in response to an automation executable creating a new instance of its Application object, it starts up without showing any windows. You can automate Word in this invisible state when you want to automate Word without confusing the user by bringing up the Word window. For this example, we want to make Word show its main window, and we do so by adding this line of code:

```csharp
theApplication.Visible = true;
```

Next, we want to create a new empty Word document into which we will generate our table. We do this by calling the Add method on the Documents collection returned by Word’s Application object. The Add method takes four optional parameters that we want to omit. Optional parameters in Word methods are specified as omitted by passing by reference a variable containing the special value `Type.Missing`. We declare a variable called `missing` that we set to `Type.Missing` and pass it by reference to each parameter we want to omit, as shown here:

```csharp
object missing = Type.Missing;
    ref missing, ref missing, ref missing, ref missing);
```

With a document created, we want to read the input text file specified by the command-line argument passed to our console application. We want to parse that text file to calculate the number of columns and rows. When we know the number of columns and rows, we use the following line of code to get a Range object from the Document object. By passing our `missing` variable to the optional parameters, the Range method will return a range that includes the entire text of the document.
We then use our Range object to add a table by calling the Add method of the Tables collection returned by the Range object. We pass the Range object again as the first parameter to the Add method to specify that we want to replace the entire contents of the document with the table. We also specify the number of rows and columns we want:

```csharp
Word.Table table = range.Tables.Add(range, rowCount, columnCount, ref missing, ref missing);
```

The Table object has a Cell method that takes a row and column and returns a Cell object. The Cell object has a Range property that returns a Range object for the cell in question that we can use to set the text and formatting of the cell. The code that sets the cells of the table is shown here. Note that as in most of the Office object models, the indices are 1-based, meaning they start with 1 as the minimum value rather than being 0-based and starting with 0 as the minimum value:

```csharp
for (columnIndex = 1; columnIndex <= columnCount; columnIndex++)
{
    Word.Cell cell = table.Cell(rowIndex, columnIndex);
    cell.Range.Text = splitRow[columnIndex];
}
```

Code to set the formatting of the table by setting the table to size to fit contents and bolding the header row is shown below. We use the Row object returned by `table.Rows[1]`, which also has a Range property that returns a Range object for the row in question. Also, we encounter code that sets the first row of the table to be bolded. One would expect to be able to write the code `table.Rows[1].Range.Bold = true`, but Word’s object model expects an int value (0 for false and 1 for true) rather than a bool. The Bold property doesn’t return a bool because the range of text could be all bold, all not bold, or partially bold. Word uses the enumerated constant `WdConstants.WdUndefined` to specify the partially bold case.

```csharp
// Format table
table.Rows[1].Range.Bold = 1;
table.AutoFitBehavior(Word.WdAutoFitBehavior.wdAutoFitContent);
```
Finally, some code at the end of the program forces Word to quit without saving changes:

```csharp
// Quit without saving changes
object saveChanges = false;
theApplication.Quit(ref saveChanges, ref missing, ref missing);
```

If you do not write this code, Word will stay running even after the console application exits. When you show the Word window by setting the Application object’s Visible property to `true`, Word puts the lifetime of the application in the hands of the end user rather than the automating program. So even when the automation executable exits, Word continues running. To force Word to exit, you must call the Quit method on Word’s Application object. If this program didn’t make the Word window visible—say for example, it created the document with the table and then saved it to a file all without showing the Word window—it would not have to call Quit because Word would exit when the program exited and released all its references to the Word objects.

To run the console application in Listing 2-3, you must create a text file that contains the text in Listing 2-2. Then pass the filename of the text file as a command-line argument to the console application. You can set up the debugger to do this by right-clicking the WordWiki project in Solution Explorer and choosing Properties. Then click the Debug tab and set the Command line arguments field to the name of your text file.

**Listing 2-3  The Complete WordWiki Implementation**

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using Office = Microsoft.Office.Core;
using Word = Microsoft.Office.Interop.Word;

namespace WordWiki
{
    class Program
    {
        static void Main(string[] args)
        {
            Word.Application theApplication = new Word.Application();
            theApplication.Visible = true;
```
object missing = System.Type.Missing;
    ref missing, ref missing, ref missing, ref missing);

TextReader reader = new System.IO.StreamReader(args[0]);

string[] separators = new string[1];
separators[0] = "||";
int rowCount = 0;
int columnCount = 0;

// Read rows and calculate number of rows and columns System.Collections.Generic.List<string> rowList =
    new System.Collections.Generic.List<string>();

string row = reader.ReadLine();
while (row != null)
{
    rowCount++;
    rowList.Add(row);
    // If this is the first row,
    // calculate the number of columns
    if (rowCount == 1)
    {
        string[] splitHeaderRow = row.Split(
            separators, StringSplitOptions.None);

        // Ignore the first and last separator
        columnCount = splitHeaderRow.Length - 2;
    }
    row = reader.ReadLine();
}

// Create a table Word.Range range = theDocument.Range(ref missing,
    ref missing);
Word.Table table = range.Tables.Add(range, rowCount,
    columnCount, ref missing, ref missing);

// Populate table int columnIndex = 1;
int rowIndex = 1;
foreach (string r in rowList)
{
Office Add-Ins

The second pattern used in Office development is the add-in pattern. This book covers several types of Office add-ins. These include VSTO add-ins for Outlook, COM add-ins for Excel and Word, and automation add-ins for Excel:

- **VSTO add-ins for Outlook**—This new VSTO feature makes it extremely easy to create an add-in for Outlook 2003. The model is the most “.NET” of all the add-in models and is very similar to the VSTO code behind model for documents. Chapter 24, “Creating Outlook Add-Ins with VSTO,” describes this model in detail.
• **COM add-ins for Excel and Word**—A C# class in a class library project can implement the IDTExtensibility2 interface and register in the registry as a COM object and COM add-in. Through COM interop, Office creates the C# class and talks to it. Chapter 23, “Developing COM Add-Ins for Word and Excel,” describes the creation of COM add-ins and some issues that make COM add-in development problematic.

• **Automation add-ins for Excel**—These managed classes expose public functions that Excel can use in formulas. The C# class must register in the registry as a COM object. Through COM interop, Excel can create an automation add-in and use its public methods in formulas. Automation add-ins and their use in Excel formulas are discussed in Chapter 3, “Programming Excel.”

This book does not discuss some Office add-in technologies. Smart Documents add-ins are not discussed because VSTO provides a much easier way of accessing Smart Document functionality, albeit at the document or template level rather than at the application level. For more information on VSTO’s support for Smart Documents, see Chapter 15, “Working with Actions Pane.”

### Creating an Outlook Add-In in VSTO

To create an Outlook add-in project in VSTO, choose Project from the New menu of the File menu in Visual Studio. Select the Visual C# node from the list of project types, and select the Office node under the Visual C# node. The Outlook add-in project appears in the list of templates. Type a name for your new Outlook add-in project, and pick a location for the project. Then click the OK button.

VSTO creates a project with references to the Outlook 2003 PIA, the core Office PIA, and other needed references, as shown in Figure 2-6. VSTO also adds a project item to the project called ThisApplication.cs. This project item contains a C# class that you will add to when implementing your Outlook add-in.
VISUAL STUDIO TOOLS FOR OFFICE

Figure 2-5 Creating a new Outlook add-in project.

Figure 2-6 The Outlook add-in project in Solution Explorer.
If you double-click the ThisApplication.cs project item, you will see the code shown in Listing 2-4. There is a simple Startup and Shutdown event handler where you can write code that executes on the startup and shutdown of the add-in. The ThisApplication class derives from an aggregate of the Outlook Application object. This allows you to access properties and methods of the Outlook Application object by writing code such as `this.Inspectors.Count` in the ThisApplication class.

Listing 2-4 The Initial Code in the ThisApplication Class in an Outlook Add-In Project

```csharp
using System;
using System.Windows.Forms;
using Outlook = Microsoft.Office.Interop.Outlook;

namespace OutlookAddin1
{
    public partial class ThisApplication
    {
        private void ThisApplication_Startup(object sender, EventArgs e)
        {
        }

        private void ThisApplication_Shutdown(object sender, EventArgs e)
        {
        }

        #region VSTO Designer generated code
        private void InternalStartup()
        {
            this.Startup += new System.EventHandler(ThisApplication_Startup);
            this.Shutdown += new System.EventHandler(ThisApplication_Shutdown);
        }
        #endregion
    }
}
```

Looking at Listing 2-4, you might wonder about the use of `partial` in the class definition. VSTO uses partial classes, which are a new feature of .NET that enables you to define part of a class in one file and another part of a class in a second file and then compile them together as one class. VSTO uses this feature to hide some additional generated code associated with the ThisApplication class from you to reduce the
complexity of the class where you write your code. The final ThisApplication class
will be compiled from the partial class in Listing 2-4 and additional code in a par-
tial class generated by VSTO that is hidden from you.

The InternalStartup method is generated by VSTO and used to hook up any event
handlers generated by VSTO. This is where the Startup and Shutdown event han-
dlers are hooked up. You should not edit this section of the code. We may omit this
block of code in some of the listings in this book, but the block of code must be in
the class—otherwise, the class will fail to compile.

We are going to add to the code in Listing 2-4 to create an add-in that will solve
an annoying problem—people replying inadvertently to an e-mail sent out to a mail-
ing alias that contains a large number of people. Unless you have “Vice President” in
your title, you probably do not want to be sending e-mail to more than, say, 25 peo-
ple at any given time. We are going to create an add-in that will warn you if you do
this and give you the “This is a potentially career-limiting move. Are you sure you
want to send this e-mail to 25,000 people?” message.

Outlook’s Application object has an ItemSend event that is raised whenever a
user sends an e-mail. We will add additional code to the Startup method of the This-
Application class to connect an event handler for the ItemSend event, as shown in
Listing 2-5. Because the ThisApplication class derives from an aggregate of Out-
look’s Application object, we can write the code this.ItemSend because ItemSend
is an event raised by the ThisApplication base class. The ItemSend event handler
takes an object parameter called item, which is the Outlook item being sent.
Because item could be any of a number of things, such as a meeting request or an
e-mail message, item is passed as an object rather than as a specific type. The Item-
Send event handler also has a bool parameter passed by reference called cancel
that can be set to true to prevent the Outlook item from being sent.

In our ItemSend event handler, we need to check to see whether the item param-
eter which is passed as an object is actually an e-mail. The easiest way to achieve
this is to use the as keyword to try to cast the item parameter to an
Outlook.MailItem. If the cast succeeds, the resulting value will be non-null, and we
will know that the item being sent is an Outlook.MailItem and therefore an e-mail
message. We can then iterate through the Recipients collection on the MailItem object
and check to see whether we are sending to any recipient lists that include more than
25 people. Each Recipient object in the Recipients collection has an AddressEntry
property that returns an AddressEntry object. The AddressEntry object has a Members property that returns a collection that we can check the count of. If we find the count to be more than 25, we will show a dialog and ask the user if she really wants to send the mail. If the user clicks the No button, we will set the cancel parameter of the ItemSend event to true to cancel the sending of career-limiting e-mail.

Listing 2-5 A VSTO Outlook Add-In That Handles the ItemSend Event and Checks for More Than 25 Recipients

```csharp
using System;
using System.Windows.Forms;
using Outlook = Microsoft.Office.Interop.Outlook;

namespace OutlookAddin1
{
    public partial class ThisApplication
    {
        private void ThisApplication_Startup(object sender, EventArgs e)
        {
            this.ItemSend += new Outlook.ApplicationEvents_11_ItemSendEventHandler(
                ThisApplication_ItemSend);
        }

        void ThisApplication_ItemSend(object item, ref bool cancel)
        {
            Outlook.MailItem myItem = item as Outlook.MailItem;

            if (myItem != null)
            {
                foreach (Outlook.Recipient recip in myItem.Recipients)
                {
                    if (recip.AddressEntry.Members.Count > 25)
                    {
                        // Ask the user if she really wants to send this e-mail
                        string message = "Send mail to {0} with {1} people?";
                        string caption = "More than 25 recipients";
                        MessageBoxButtons buttons = MessageBoxButtons.YesNo;
                        DialogResult result;

                        result = MessageBox.Show(String.Format(message,
                            recip.AddressEntry.Name,
                            recip.AddressEntry.Members.Count),
                            caption, buttons);
                    }
                }
            }
        }
    }
}
```
When you run the project with the code shown in Listing 2-4, Outlook launches and the add-in loads. Try sending a mail to an alias that includes more than 25 people—you might want to go offline first in case you mistyped the code. If all works right, the add-in will display a dialog box warning you that you are sending an e-mail to more than 25 people, and you will be able to cancel the send of the e-mail. Exit Outlook to end your debugging session.

Chapter 24, “Creating Outlook Add-Ins with VSTO,” discusses VSTO Outlook add-ins in more detail. Chapters 9 through 11—“Programming Outlook,” “Working with Outlook Events,” and “Working with Outlook Objects,” respectively—discuss the Outlook object model.
Code Behind a Document

VSTO supports code behind a document by requiring that the developer use classes generated in a VSTO project that have pre-hooked-up context and pre-hooked-up events. These classes are sometimes called “code behind” classes because they are code associated with a particular document or worksheet. In Word, there is one code behind class corresponding to the document. In Excel, there are multiple code behind classes—one for the workbook and one for each worksheet or chart sheet in the workbook.

The first time your code runs in a VSTO code behind the document project is when Office raises the Startup event handled by any of the code behind classes created for you. VSTO provides context via the base class of the code behind class you are writing code in. A VSTO code behind class customizing an Excel worksheet derives from a base class that contains all the methods, properties, and events of an Excel worksheet. This enables you to write code such as this in the Startup method of a worksheet class.

```csharp
MessageBox.Show(String.Format("{0} is the sheet name", this.Name));
```

By using `this.Name`, you are referring to the Name property of the Excel Worksheet object inherited from the base class. Listing 2-6 shows a VSTO code behind class for an Excel Worksheet. In addition to the Startup and Shutdown methods in the code behind class, there is also a generated method called InternalStartup. You should not put any of your code in this InternalStartup method because it is auto-generated by VSTO and modifying it can break Visual Studio’s support for code behind classes. Instead, your startup code should go in the Startup event handler. VSTO code behind document classes also use partial classes to hide some additional code generated by VSTO.

**Listing 2-6 A VSTO Excel Workbook Customization**

```csharp
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;
```
namespace ExcelWorkbook1
{
    public partial class Sheet1
    {
        private void Sheet1_Startup(object sender, EventArgs e)
        {
            // Initial entry point.
            // This code gets run first when the code behind is created
            // The context is implicit in the Sheet1 class
            MessageBox.Show("Code behind the document running.");
            MessageBox.Show(String.Format("{0} is the sheet name", this.Name));
        }

        private void Sheet1_Shutdown(object sender, EventArgs e)
        {
        }

        #region VSTO Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InternalStartup()
        {
            this.Startup += new System.EventHandler(Sheet1_Startup);
            this.Shutdown += new System.EventHandler(Sheet1_Shutdown);
        }

        #endregion
    }
}

VSTO Code Behind a Document in Excel

In this section, we create some simple code behind a document in Excel using VSTO. First, start up VSTO and choose the File > New > Project menu item. As you have seen previously, navigate to the Office node under the Visual C# root.

We will create an Excel workbook project using C#. If you already have a workbook that you want to add VSTO customization code behind, you can specify its location in the dialog box shown in Figure 2-8 that appears after you click OK in the New Project dialog. This time we will just start from scratch, creating a new, blank workbook.
Figure 2-7 Using the New Project dialog to create an Excel Workbook project.

Figure 2-8 Selecting the workbook to associate with your code behind.
Notice a few interesting things in Figure 2-9. First, Excel is running inside Visual Studio 2005 as a designer, just the same as a Windows Forms designer would when developing a Windows Forms project.

Second, look at the menu bar shown in Figure 2-10. VSTO merges the Visual Studio menus (Build, Debug, and so on) and the Excel menu items (Format, Data, and so on) together. Menu items that appear in both Visual Studio and Excel (Tools, for example) merge by adding a submenu to the Visual Studio menu, such as Microsoft Office Excel Tools, that can be selected to see the Excel Tools menu.
Third, notice in Figure 2-9 that the toolbox contains a new category: Excel Controls. When designing a document using Visual Studio, you can create named ranges and list objects using the Excel menu items familiar to Excel users, or the toolbox idiom familiar to Visual Studio users.

Fourth, notice that the Properties window shows the properties of the selected object—in this case, Sheet1. You can use the Properties window to edit properties of Excel’s objects the same way that you would edit properties of controls and forms in a Windows Forms project.

Fifth, notice that the Solution Explorer has four classes in it already. Each underlying Excel Worksheet and Workbook object is represented by a .NET class that you can extend and customize. As you make changes to the document in the designer, the code behind updates automatically. For example, drag a list object from the toolbox onto the Sheet1 designer, and draw it to be 10 rows by 4 columns, as shown in Figure 2-11.
As you can see from the Properties window, the designer has chosen a default name for the new list object. We could edit it, but in this example, we will keep the default name List1.

Let's take a look at the code behind this worksheet and make some simple changes to it. Right-click Sheet1.cs in the Solution Explorer and choose View Code. We are going to briefly illustrate two VSTO features: ActionsPane and list object data binding. We will declare a Windows Forms button as a member variable of the class and call it myButton. In the Startup event, we will show that button in the Document Actions task pane of Excel by adding it to the ActionsPane’s Controls collection. Doing so will cause Excel to show the Document Actions task pane and display our
button. We will also handle the Click event of the button, and when the button is clicked we will data bind our list object to a randomly generated DataTable. Listing 2-7 shows this code.

**Listing 2-7 A VSTO Customization That Adds a Control to the Document Actions Task Pane and Data Binds a ListObject Control to a DataTable**

```csharp
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace ExcelWorkbook1
{
    public partial class Sheet1
    {
        Button myButton = new Button();
        DataTable table;

        private void Sheet1_Startup(object sender, EventArgs e)
        {
            myButton.Text = "Databind!";
            myButton.Click += new EventHandler(myButton_Click);
            Globals.ThisWorkbook.ActionsPane.Controls.Add(myButton);
        }

        void myButton_Click(object sender, EventArgs e)
        {
            List1.DataSource = null;
            table = new DataTable();
            Random r = new Random();

            for (int i = 0; i < 4; i++)
                table.Columns.Add("Col " + i.ToString());

            for (int i = 0; i < 20; i++)
                table.Rows.Add(r.NextDouble(), r.NextDouble(),
                                r.NextDouble(), r.NextDouble());

            List1.DataSource = table;
        }
    }
}
```
private void Sheet1_Shutdown(object sender, EventArgs e) {
}

#region VSTO Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InternalStartup()
{
    this.Startup += new System.EventHandler(Sheet1_Startup);
    this.Shutdown += new System.EventHandler(Sheet1_Shutdown);
}
#endregion

Build and run the code, and sure enough Excel starts up, the Startup event is raised for the sheet, and the button is added to the actions pane. Click the button and a random DataTable is generated and bound to the list object, as shown in Figure 2-12. Exit Excel to end your debugging session.

We have briefly illustrated VSTO’s support for the Document Actions task pane and the ability to data bind that VSTO adds to Excel’s list object. For more information on VSTO’s support for the Document Actions task pane, see Chapter 15, “Working with Actions Pane.” For more information on VSTO’s support for data binding, see Chapter 17, “VSTO Data Programming.”
Conclusion

This chapter introduced the three basic patterns of Office solutions: an automation executable, an add-in, and code behind a document. The chapter also introduced how to build solutions following these three basic patterns using Visual Studio 2005 and Visual Studio 2005 Tools for Office.

Now that you know how to create a basic automation executable, add-in, and code behind the document solution, you will use these skills in the next chapters as the focus turns to specific functionality of Excel, Word, Outlook, and InfoPath that you can use in your solutions.

Figure 2-12 The result of running Listing 2-7 and clicking the button we added to the Document Actions task pane.
This chapter has only served as an introduction to add-ins and code behind documents. Chapter 24 covers VSTO add-ins for Outlook. Chapter 23 covers COM add-ins for Word and Excel. Chapter 3 covers automation add-ins for Excel. Chapters 13 through 17 cover the code behind document model of VSTO in detail.