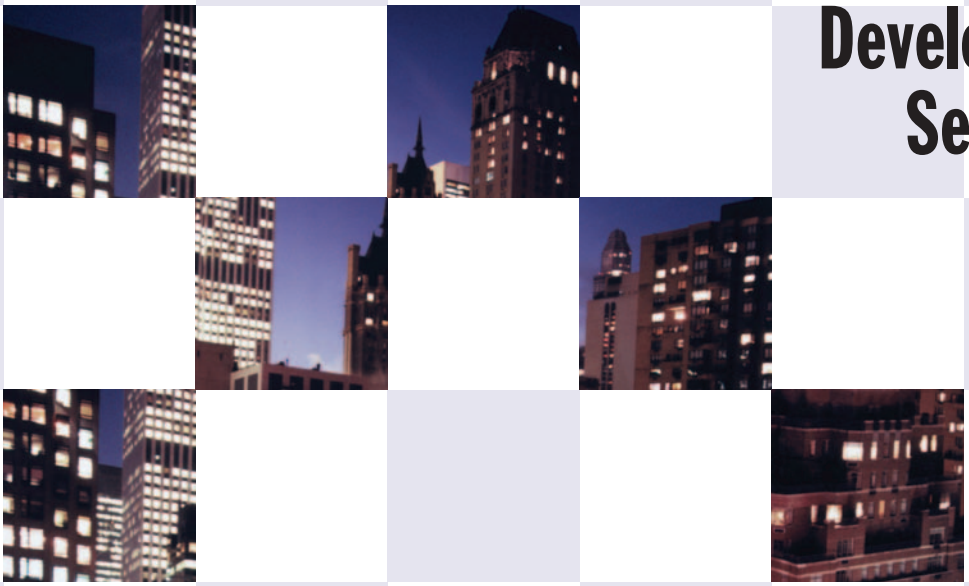
FREE CHAPTER
BOOK AVAILABLE SEPTEMBER 2005

Visual Studio Tools for Office

Using C# with Word, Excel, Outlook, and InfoPath

Microsoft® .net™ Development Series



Eric Carter
Eric Lippert

DRAFT MANUSCRIPT

Books Available

September 2005

This manuscript has been provided by Pearson Education at this early stage to create awareness for this upcoming book. **It has not been copyedited or proofread yet**; we trust that you will judge this book on technical merit, not on grammatical and punctuation errors that will be fixed at a later stage.

Table of Contents

Chapter 1: Introduction to Office Programming

- Why Office Programming?
  - Office Programming and the Professional Developer
  - Why .NET for Office?
  - How .NET is it?
- Office Object Models
  - Objects
  - Collections
  - Enumerations
- Properties, Methods, and Events
  - Properties
    - Parameterized Properties
    - Properties Common to Most Objects
  - Methods
    - Optional Parameters in Word
  - Events
    - The “My Button Stopped Working” Issue
    - When Method Names and Event Names Collide
- The Office Primary Interop Assemblies
  - Installing the Primary Interop Assemblies
  - Referencing the Primary Interop Assemblies
  - Browsing the Primary Interop Assemblies
    - Dummy Methods
- Conclusion

Chapter 2: Introduction to Office Solutions

- The Three Basic Patterns of Office Solutions
  - Hosted Code
    - Discovery of Hosted Code
    - Context Provided to Hosted Code
    - Entry Point for Hosted Code
  - How Code Gets Run After Startup
    - Code Runs In Response to Events Fired By Office
    - Interface Methods Called On Objects Provided To Office
    - Events Raised on Code Behind Classes
  - How Code Gets Unloaded
- Office Automation Executables
  - Creating a Console Application that Automates Word
- Office Add-Ins
  - Creating an Outlook Add-in in VSTO
- Code behind a Document
  - VSTO 2005 Code behind a Document in Excel
- Conclusion

Chapter 3: Programming in Excel

- Ways to Customize Excel
  - Automation Executable
  - Add-Ins
  - VSTO 2005 Code Behind
  - Smart Documents and XML Expansion Packs
  - Smart Tags
  - Server-Generated Documents
  - Research Services
- User Defined Functions
  - Building a Managed Automation Add-in that provides User Defined Functions
  - Using Your Managed Automation Add-in in Excel

    - Some Additional User Defined Functions
    - Debugging User Defined Functions in a Managed Automation Add-ins
    - Deploying Managed Automation Add-ins
- Introduction to the Excel Object Model
    - The Object Hierarchy
- Conclusion

Chapter 4: Working with Excel Events

- Events in the Excel Object Model
        - New Workbook and Worksheet Events
        - Activation and Deactivation Events
        - Double Click and Right Click Events
        - Calculate Events
        - Change Events
        - Follow Hyperlink Events
        - Selection Change Events
        - Window Resize Events
        - Add-In Install and Uninstall Events
        - XML Import and Export Events
        - Before Close Events
        - Before Print Events
        - Before Save Events
        - Open Events
        - Toolbar and Menu Events
        - Additional Events
    - Events in Visual Studio Tools for Office 2005
- Conclusion

Chapter 5: Working with Excel Objects

- The Application Object
    - Controlling Excel's Screen Updating Behavior
    - Controlling the Dialogs and Alerts that Excel Displays
    - Changing the Mouse Pointer
    - Displaying a Message in Excel's Status Bar
    - Controlling User Interaction with Excel While Your Code Is Running
    - A Property You Should Never Use
    - Controlling the Editing Experience in Excel
        - Controlling the look of Excel
    - Controlling file and printer settings
    - Properties That Return Active or Selected Objects
    - Properties That Return Important Collections
    - Controlling the Calculation of Workbooks
    - Using Built In Excel Functions in Your Code
        - Selecting and Activating a Range of Cells
    - Spell Checking
    - Sending a Workbook in E-Mail
    - Quitting Excel
    - Undo in Excel
    - Sending Keyboard Commands to Excel
- Workbooks Collection
    - Enumerating Over the Open Workbooks
    - Accessing a Workbook in the Workbooks Collection
    - Creating a New Workbook
    - Opening an Existing Workbook
    - Closing All the Open Workbooks
- Workbook Object
    - Properties that Return Active or Selected Objects
    - Properties That Return Important Collections
    - Accessing Document Properties
    - Saving an Excel Workbook
    - Naming Ranges of Cells

        - When Excel Is Embedded In Another Application
        - Creating and Activating Windows
        - Printing a Workbook
        - Protecting a Workbook
- Worksheets, Charts, and Sheets Collection
    - Enumerating Over the Open Sheets
    - Accessing a Sheet in the Collection
    - Adding a Worksheet or Chart Sheet
    - Copying a Sheet
    - Moving a Sheet
- DocumentProperties Collection and DocumentProperty Object
    - Enumerating Over The DocumentProperty objects
    - Accessing a DocumentProperty in the DocumentProperties Collection
    - Adding a DocumentProperty
- Windows Collection
    - Enumerating Over the Open Windows
    - Accessing a Window in the Collection
    - Arranging Windows
- Window Object
    - Positioning a Window
    - Display Settings Associated with a Window
- Names Collection and Name Object
    - Enumerating Over the Names Collection
    - Accessing a Name in the Names Collection
    - The Name Object
- Worksheet Object
    - Worksheet Management
    - Working with Names
    - Working With Worksheet Custom Properties
    - Protecting a Worksheet
    - Working with OLEObjects
    - Working with Shapes
    - Working with ChartObjects
    - Working with Lists
- Range Object

        - Getting a Range Object for a Particular Cell or Range of Cells
        - Working with Addresses
        - Creating New Ranges Using Operator Methods
        - Working with Areas
        - Working with Cells
        - Working with Rows and Columns
        - Working with Regions
        - Selecting a Range
        - Editing the Values in a Range
        - Copying, Clearing, and Deleting Ranges
        - Finding Text in a Range
- Special Excel Issues
    - The Excel Locale Issue
        - Using DateTime for Dates
        - Switching the Locale to English and Back
        - Old Format or Invalid Type Library Error
    - Converting Excel Dates to DateTime
- Conclusion

Chapter 6: Programming in Word

- Ways to Customize Word
    - Automation Executable
    - Add-Ins
    - VSTO 2005 Code Behind
    - Smart Documents and XML Expansion Packs
    - Smart Tags
    - Server Generated Documents
    - Research Services
        - Getting Started with Research Services
        - A Simple Research Service
        - Registering the Research Service with Word
        - Using the Research Service
        - More Research Service Resources

- Introduction to the Word Object Model
    - The Object Hierarchy
- Conclusion

Chapter 7: Working with Word Events

- Events in the Word Object Model
    - Why are there multiple Application and Document Event Interfaces?
    - Visual Studio Generation of Event Handlers
    - Startup and Shutdown Events
    - New and Open Document Events
    - Document Close Events
    - Document Save Events
    - Document Activation Events
    - Document Print Events
    - Mouse Events
    - Selection Events
    - Window Sizing Events
    - XML Events
    - Sync Events
    - EPostage Events
    - Mail Merge Events
    - CommandBar Events
    - Events in Visual Studio Tools for Office 2005
- Conclusion

Chapter 8: Working with Word Objects

- The Application Object
    - Controlling Word's Screen Updating Behavior
    - Controlling the Dialogs and Alerts that Word Displays
    - Changing the Mouse Pointer
    - Displaying a Message in Word's Status Bar or Window Caption

- Controlling the look of Word
- Properties That Return Active or Selected Objects
- Properties That Return Important Collections
- Accessing Items in Collections
- Navigating a Document
- Working with Word's Options
- Working with the New and Getting Started Document Task Pane
- Working with the File Save Format Options
- Working with File Dialogs
- User Information
- Checking Grammar and Spelling
- Exiting Word

- Working with the Dialog Object
  - Showing the Dialog and Letting Word Execute Actions
  - Selecting the Tab on a Dialog box
  - Showing the Dialog and Preventing Word from Executing Actions
  - Getting and Setting Fields in a Dialog
- Working with Windows
  - Creating New Windows
  - Enumerating Over the Open Windows
  - Accessing a Window in the Collection
  - Arranging Windows
- Working with Templates
  - Types of Templates
- Working with Documents
  - Enumerating Over the Open Documents
  - Accessing a Document in the Documents Collection
  - Creating a New Document
  - Opening an Existing Document
  - Closing All Open Documents
  - Saving All Open Documents
- Working with a Document
  - Preserving the Dirty State of a Document
  - Closing and Saving a Document

        - Working with Windows Associated With a Document
        - Changing the Template Attached to a Document
        - Important Collections associated with both Document and Range
        - Important Collections associated with Document Only
        - Working with Document Properties
        - Checking Spelling and Grammar in Documents and Ranges
        - Printing a Document
        - Working with Document Protection
        - Working with Password Protection
        - Undo and Redo
- Working with the Range Object
        - Getting a Range
        - Identifying a Range
        - Changing a Range
        - Moving a Range
        - Ranges and Stories
        - Navigating a Range
        - Collapsing a Range
        - Getting Text from a Range
        - Setting the Text in a Range
        - Inserting Non-printing Characters and Breaks
        - Working with Formatting
        - Find and Replace
- Working with Bookmarks
- Working with Tables
- Conclusion

Chapter 9: Programming in Outlook

- Ways to Customize Outlook
    - Automation Executable
    - Add-Ins
    - Outlook Add-In Issues

        - Getting Outlook to call OnDisconnection on your Add-in and Shut Down
        - Understanding RCWs, Application Domains, and Why to Avoid Calling ReleaseComObject
        - How Outlook Add-in Development Should Be
    - Smart Tags
        - Smart Tags when Word is the E-mail Editor
        - Smart Tags in the Research Task Pane when Word is the E-mail Editor
        - Smart Tags Embedded in HTML Format E-Mail and Displayed in the Reading Pane
        - Persona Menu Smart Tags
    - Custom Property Pages
- Introduction to the Outlook Object Model
    - The Object Hierarchy of the Outlook Object Model
- Introduction to the Collaboration Data Objects
- Conclusion

Chapter 10: Working with Outlook Events

- Events in the Outlook Object Model
    - Why are there multiple Event Interfaces?
- Application Level Events
    - Startup and Quit Events
    - Activation Events
    - New Window Events
    - Window Events
    - Close Events
    - View and Selection Change Events
    - Folder Change Events
- Outlook item Events
    - Item Addition, Deletion, and Change Events
    - Copy, Paste, Cut, and Delete Events
    - Property Change Events
    - Open, Read, Write, and Close Events
    - E-Mail Events

        - Attachment Events
        - Custom Action Events
    - Other Events
    - Conclusion

Chapter 11: Working with Outlook Objects

- The Application Object
    - Methods and Properties That Return Active or Selected Objects
    - Properties That Return Important Collections
    - Performing a Search and Creating a Search Folder
    - Copying a File into an Outlook Folder
    - Quitting Outlook
- Working with the Explorers and Inspectors Collections
- The Explorer Object
    - Working with the Selected Folder and View, and Selected Items
    - Working With an Explorer Window
    - Adding Buttons and Menus to an Explorer Window
    - Associating a Web View With a Folder
- The Inspector Object
    - Working with the Outlook Item Associated with the Inspector
    - Working With an Inspector Window
    - Working with Different Inspector Editor Types
    - Adding Buttons and Menus to an Inspector Window
- The NameSpace Object
    - Working With the Root Folders of the Open Outlook Stores
    - Adding and Removing Outlook Stores
    - Determining the Current User
    - Checking if Outlook is Offline
    - Getting Standard Folders like the Inbox Folder
    - Getting a Folder or Outlook Item by ID
    - Accessing Address Books and Address Entries

        - Displaying the Outlook Folder Picker Dialog
- The MAPIFolder Object
    - Other Identifiers for a Folder
    - Accessing SubFolders contained in a Folder
    - Accessing Items contained in a Folder
    - Working with a Folder's View Settings
    - Copying or Moving a Folder to a new location
    - Displaying a Folder in an Explorer View
- The Items Collection
    - Iterating over Outlook Items
    - Finding an Outlook Items
    - Adding an Outlook Item to an Items Collection
- Properties and Methods Common to Outlook Items
    - Creating an Outlook Item
    - Identifying the Specific Type of an Outlook Item
    - Other Properties Associated With All Outlook Items
    - Copying or Moving an Outlook Item to a New Location
    - Deleting an Outlook Item
    - Displaying an Outlook Item in an Inspector View
    - Working with Built-In and Custom Properties Associated With an Outlook Item
    - Saving an Outlook Item
    - Showing the Categories Dialog for an Outlook Item
    - Mail Properties and Methods
- Outlook Issues
    - Outlook Object Model Security
    - Extended MAPI
- Conclusion

Chapter 12: Programming in InfoPath

Chapter 13: VSTO View Programming

- VSTO 2005 Extensions To Word and Excel View Objects
  - Aggregation, Inheritance and Implementation
  - Hooking Up The Aggregates
  - Obtaining the Aggregated Object
  - Aggregation and Windows Forms Controls
  - Improving C# Interoperability
  - The “Tag” Field
  - Event Model Improvements
- Dynamic Controls
  - The Controls Collection
  - Enumerating and Searching the Collection
  - Adding New Windows Forms Controls Dynamically
  - Adding New Word and Excel View Controls Dynamically
  - Removing Controls
  - Dynamic Controls Information Is Not Persisted
- Advanced Topic: Dynamic Views
- Advanced Topic: Inspecting the Generated Code
  - The Startup and Shutdown Sequences
  - The Globals Class in Excel
- VSTO 2005 Extensions to the Word and Excel Object Models
  - The Word Document View Class
  - The Word Bookmark View Control
  - The Word XMLNode and XMLNodes View Classes
  - The Excel Workbook View Class
  - The Excel Worksheet View Class
  - The Excel Chartsheet View Class And Chart View Control
  - The Excel NamedRange, XmlMappedRange and ListObject View Controls

Chapter 14: Using Windows Forms in VSTO

- Introduction to Windows Forms in Visual Studio Tools for Office
    - Moving from ActiveX to Windows Forms
- Why Use Windows Forms Controls On the Document Surface?
- Windows Form control implementation in Office 2003
- Limitations of hosting a control on a document
- Adding controls to your document
    - Insertion behavior in Word
    - Insertion behavior in Excel
- Working with the control in the designer
    - Excel Control Extender Properties that you can use on the control
    - Word Control Properties
    - Layout of controls on the document or worksheet surface
- Writing code behind the control
    - Events that never fire for a control
- Adding controls at runtime
    - What exactly does the Controls collection contain?
    - Dynamic versus Static
- Using the Controls collection
    - Deleting controls at runtime
- Conclusion

Chapter 15: Working with the Actions Pane

- Introduction to the Actions Pane
    - What About Smart Documents?
- Working with the ActionsPane Control
    - ActionsPane Architecture
    - Adding Windows Forms Controls to the ActionsPane
    - Adding a Custom User Control to the ActionsPane
    - Contextually Changing the ActionsPane

        - Detecting the Orientation of the ActionsPane
        - Scrolling the ActionsPane
        - Showing and Hiding the ActionsPane
        - Attaching and Detaching the ActionsPane
        - Some Methods & Properties to Avoid
- Conclusion

Chapter 16: Working with Smart Tags in VSTO

- Introduction to Smart Tags
    - Configuring Smart Tags in Word and Excel
    - The Persistent Tagging Generated by Smart Tags
- Document Level Smart Tags with VSTO
    - Action Events
    - Using Varying Numbers of Terms
    - Using Regular Expressions
    - Having a Varying Number of Actions
    - Creating a Custom Smart Tag Class
- Application Level Smart Tags
- Conclusion

Chapter 17: VSTO Data Programming

- Whence DataSets?
    - A Disconnected Strategy
    - Rolling Your Own DataSet Filling Code
    - Working With Many Tables
- Typed DataSets
- Creating Data-Bound View Controls With The Designer
    - Creating DataBound Excel Documents With ListObject and XmlMappedRange
    - Complex and Simple Data Binding
    - Data Binding In Word Documents
- ADO.NET Data Binding: Looking Behind The Scenes
    - Binding Managers Manage Currency

    - Using Binding Sources as Proxies
- Binding-Related Extensions To View and View Control Classes
    - Extensions to the List Object View Control in Excel
        - New Data-related List Object View Control Properties and Methods:
        - New Data-related List Object Events:
        - New Exception:
- Master-Detail Data Binding
- Conclusion

Chapter 18: Server Data Scenarios

- Server Data Scenarios
    - Data Bound VSTO Documents
    - XML File Formats
    - Accessing the Data Island
- Using ServerDocument and ASP.NET
    - Setting up the server
    - An alternate approach: Create a custom handler
- The ServerDocument Object Can Read Application Information
- The ServerDocument Object Model
    - Loading Documents With Constructors
    - Saving and Closing Documents
    - Static helper methods
    - Application Manifest objects, methods and properties
    - Cached Data objects, methods and properties
    - Be Careful
- Conclusion

Chapter 19: .NET Code Security

- Code Access Security vs. Role-Based Security
- Code Access Security in .NET
    - The Machine Policy Level
    - Kinds of Evidence
    - Combining Policy Levels
    - The User Policy Level
    - Full Trust and Partial Trust
    - The VSTO Application Domain Policy Level
    - Resolving VSTO Policy
- Location, Location, Location
- Strong Names
    - Creating A Strong Name Code Group
    - How Strong Names Are Implemented
    - Why Create A Child Code Group?
    - Implementing Strong-Named Assemblies
        - Designate a Signing Authority
        - Create a Key Pair
        - Developers Delay-Sign the Assembly
        - Really Sign the Assembly
    - Public Keys and Public Key Tokens
- Publisher Certificates
    - License to Code
    - Obtaining Certificates
- Trusting the Document
    - Trusting Just Office Documents
        - Why Is MSOSec not in the GAC by default?
- Deploying Policy Throughout The Enterprise
- Conclusion

Chapter 20: Deployment and Updating of Office Solutions

Chapter 21: Working with XML in Excel

- Introduction to Excel's XML Features
- Introduction to XML Schema Creation in Visual Studio
- An End to End Scenario
    - Creating the Schema using Visual Studio
    - Adding a Schema to the Excel workbook
    - Mapping the Schema to the Excel workbook
- Advanced XML Features in Excel
    - Importing XML and Refresh XML Data
    - The XML Map Properties Dialog
    - XML Schema Validation
    - Data Formating And Layout
    - Appending Data to Lists
- Excel Friendly XML Schemas
    - Unsupported XML Schema Constructs
    - Constructs that can be Mapped but not Exported
    - VSTO Friendly Schemas
    - How XML Schema Data Types are Mapped To Excel Cell Formats
- VSTO Support for Excel Schema Mapping
    - XMLMappedRange Controls
    - ListObject Controls
    - Schema Added to the VSTO Project
    - Combining XML Mapping with VSTO Databinding
- Conclusion

Chapter 26: Working with XML in Word

- Introduction to Word's XML Features
- An End to End Scenario: Creating a Schema and Mapping it Into a Word Document
    - Creating the Schema using Visual Studio
    - Adding a Schema to the Word Document
    - The XML Options Dialog and Mixed Content

    - Creating a Document with Mapped XML Structure
- Exporting the Mapped XML in the Document to an XML Data File
- Importing an XML Data File into the Mapped Document
    - Creating the XSLT File
    - Manually Converting the Book Order XML file Using the XSLT file
    - Automatically Applying an XSLT File When XML Conforming to the Book Order Schema is Opened
- Advanced XML Features in Word
    - The XML Options Dialog
- VSTO Support for Word Schema Mapping
    - XMLNode Controls
    - XMLNode Control
    - XMLNodes Controls
    - Loading XML Programmatically with UpdateXml
- Conclusion

Chapter 23: Developing Outlook Add-ins with VSTO

- Introduction to Outlook Add-ins
- Creating an Outlook Add-in in VSTO
    - Security
    - Manifest Updating
    - Installing
    - Other VSTO Features
- Conclusion

Chapter 24: Developing COM Add-ins for Word and Excel

- Introduction to Add-ins
        - Outlook Add-ins
- Scenarios for using Add-ins

- How an Add-In Is Registered
  - Registry Location of an Add-in: HKEY_CURRENT_USER or HKEY_LOCAL_MACHINE
  - Registry Entries Required for an Add-in
- Implementing IDTExtensibility2
  - Startup Order
  - OnAddInsUpdate Method
  - OnBeginShutdown Method
  - OnConnection Method
  - OnDisconnection Method
  - OnStartupComplete Method
  - A Simple Implementation
- Writing an Add-in Using Visual Studio
  - Changing the Add-in Project to be More Office Specific
  - Setting the Start Action
  - My Add-in Project Doesn't Work Anymore--What Happened?
  - A Simple Word Add-in
- The Pitfalls of MsCoree.dll
  - Com Interop and REGASM.EXE
  - Mscoree.dll and Managed Add-ins
  - Problems with Using Mscoree.dll to Load your add-in
    - Problem 1: Mscoree.dll Can Be Disabled Causing All Managed Add-ins to Stop Loading
    - Problem 2: Mscoree.dll Cannot Be Signed
    - Problem 3: Mscoree.dll loads all add-ins into the same AppDomain
- Shimming: A Solution to the Problems With MsCoree.dll
    - How to build a Shim DLL
- Conclusion

Preface

Three years ago as the first release of Visual Studio .NET and the .NET Framework were nearing completion; a few of us at Microsoft realized that Office programming was going to miss the .NET wave unless we did something about it.

What had come before was Visual Basic for Applications (VBA), a simple development environment integrated into all the Office applications. Each Office application had a rich object model that was accessed via a technology known as COM. Millions of developers identified themselves as "Office developers" and used VBA and the Office COM object models to do everything from automating repetitive tasks to creating complete business solutions that leveraged the rich features and user interface of Office. These developers realized that their users were spending their days in Office. By building solutions that ran inside of Office, they not only made their users happy, but they were able to create solutions that did more and cost less by reusing functionality already available in the Office applications.

Unfortunately, because of some limitations of VBA, Office programming was starting to get a bad rap. Solutions developed in VBA by small workgroups or individuals would gain momentum and a professional developer would have to take them over and start supporting them. To a professional developer, the VBA environment felt simple and limited and of course it enforced a single language—Visual Basic. VBA embedded code in every customized document, which made it hard to fix bugs and update solutions as a bug would get replicated in documents across the enterprise. Security weaknesses in the VBA model caused by a rash of worms and macro viruses that made enterprises turn VBA off.

Visual Studio .NET and the .NET Framework provided a way to address all these problems. There was a huge opportunity to not only combine the richness of the new .NET Framework and developer tools with the powerful platform that Office has always provided for developers but to also solve the problems that were plaguing VBA. The result of this realization was Visual Studio Tools for Office or VSTO.

The first version of VSTO was simple, but it accomplished the key goal of letting professional developers use the full power of Visual Studio .NET and the .NET Framework to put code behind Excel 2003 and Word 2003 documents and templates. It let professional developers develop Office solutions in VB.NET and C#. It solved the problem of embedded code by

linking a document to a .NET assembly rather than embedding it in the document. It also introduced a new security model that used .NET code access security to prevent worms and macro viruses.

The second version of VSTO known as VSTO 2005, the version of VSTO covered by this book, is even more ambitious. It brings with it functionality never available to the Office developer before such as data binding and data/view separation, design time views of Excel and Word documents inside Visual Studio, rich support for Windows Forms controls in the document, the ability to create custom Office task panes, server side programming support against Office—and that's just scratching the surface. Although the primary target of VSTO is the professional developer, that doesn't mean that building an Office solution with VSTO is rocket science. VSTO makes it possible to create very rich applications with just a few lines of code.

This book tries to put into one place all the information you will need to be successful using VSTO to program against Word 2003, Excel 2003, Outlook 2003, and InfoPath 2003. It introduces the Office object models and covers most commonly used objects in those object models. In addition, this book will help you avoid some pitfalls that result from the COM origins of the Office object models.

This book will also give you an insider view of all the rich features of VSTO. We participated in the design and implementation of many of these features. We can therefore speak from the unique perspective of living and breathing VSTO for the past three years. Programming Office using VSTO is powerful and fun. We hope you enjoy using VSTO as much as we enjoyed writing about it and creating it.

Eric Carter
Eric Lippert
May 2005

About the authors

Eric Carter

Eric Carter is a Lead Developer on the Visual Studio Tools for Office (VSTO) team at Microsoft. He helped invent, design, and implement many of the features that are in VSTO today. Previously at Microsoft he worked on Visual Studio for Applications, the Visual Studio Macros IDE, and Visual Basic for Applications for Office 2000 and Office 2003.

Eric Lippert

Eric Lippert's primary focus during his nine years at Microsoft has been on improving the lives of developers by designing and implementing useful programming languages and development tools. He's worked on the Windows Scripting family of technologies and, most recently, Visual Studio Tools For Office.

Chapter 2
Introduction to Office Solutions

The Three Basic Patterns of Office Solutions

Now that we have considered the basic pattern of the Office object models, let us consider how developers pattern and build their Office solutions. There are three patterns that most solutions built using Office follow.

- 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 .NET 2005. A typical example is a stand-alone 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 that will in turn start 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 rather than requiring that a separate process from the Office application is running. 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 allows 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 VB.NET code behind using Visual Studio Tools for Office 2005. 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 will discuss an additional pattern 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. We will discuss this pattern in Chapter 18.

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

In order 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 registry so 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 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 may 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

Once hosted code starts up, code continues to run in one or more of the following ways.

Code Runs In Response to Events Fired 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 1-26 shows a simple class that listens to the change event that Excel's Worksheet object raises. Typically, you will hook up event listeners like the one shown in Listing 1-26 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

We now consider 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. Office 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 as 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. This is 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. Chapters 3 through 5 cover the Excel object model in more detail.

Listing 2-1: Automation of a Excel via a console application.

```
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(myExce
lApp_SheetBeforeDoubleClick);

      while (exit == false)
        System.Windows.Forms.Application.DoEvents();
    }
```

```
    static void myExcelApp_SheetBeforeDoubleClick(object
Sh, Microsoft.Office.Interop.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 a 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 on 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 on 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

In this section, we are going to walk through the creation of a simple console application that automates Word. A *wiki* is a kind of online encyclopedia that users can contribute to. For an example, see http://www.officewiki.net for a wiki that documents the Office 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 like 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.

Property or Method	Name	Return Type
Property	Application	Application

Property	Autoload	Boolean
Property	Compiled	Boolean
Property	Creator	Int32
Method	Delete	Void
Property	Index	Int32
Property	Installed	Boolean
Property	Name	String
Property	Parent	Object
Property	Path	String

Listing 2-2: A Wiki text representation of Table 1-4.

```
||Property or Method||Name||Return Type||
||Property||Application||Application||
||Property||Autoload||Boolean||
||Property||Compiled||Boolean||
||Property||Creator||Int32||
||Method||Delete||Void||
||Property||Index||Int32||
||Property||Installed||Boolean||
||Property||Name||String||
||Property||Parent||Object||
||Property||Path||String||
```

We will use Visual Studio .NET 2005 to create a console application. After launching Visual Studio, choose New Project… from the File menu. The new project dialog shows a variety of project types. Select the Visual C# node from the list of project types and select the Windows node under the Visual C# node. This is slightly counter intuitive as there is an Office node available as well, but the Office node only shows VSTO code behind document projects and the VSTO Outlook add-in project.

After you select the Windows node, you will see in the window to the right the available templates. Select the Console Application template. Name your console application project then press the OK button to create your project. In Figure 2-1 we've 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.

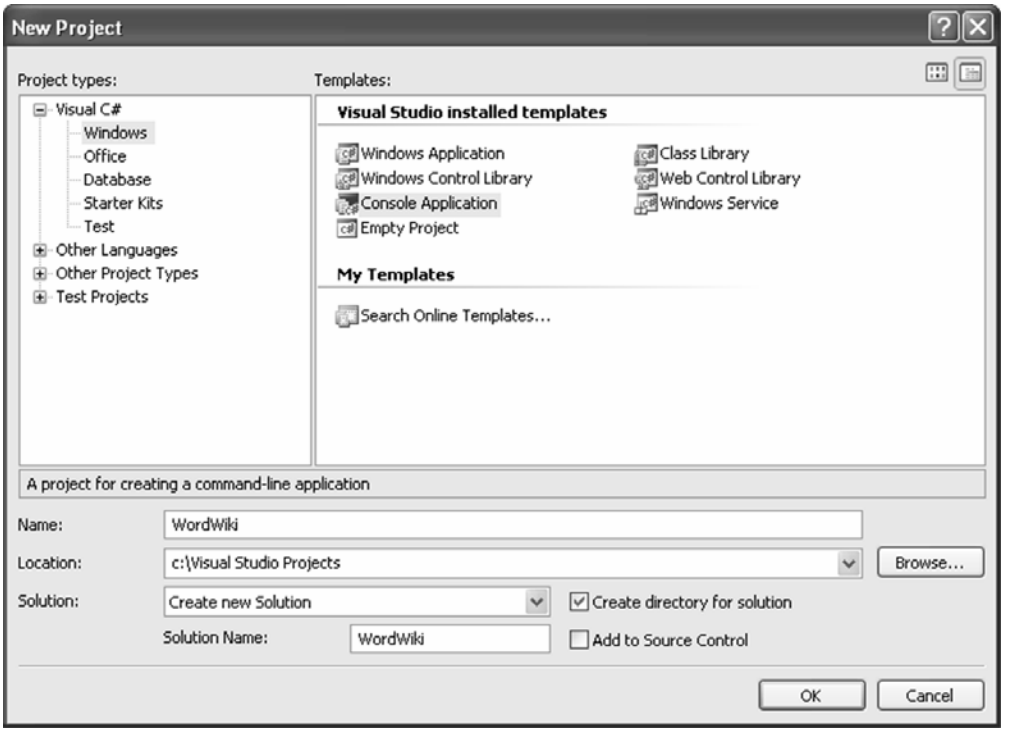


Figure 2-1*: Creating a console application from the New Project dialog.*

Once you press 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 on the References folder and choosing Add Reference… from the popup menu that appears. This shows the Add Reference dialog in Figure 2-3. Click on the COM tab and select the Microsoft Word 11.0 Object Library to add a reference to the Word 2003 PIA. Then click the OK button.

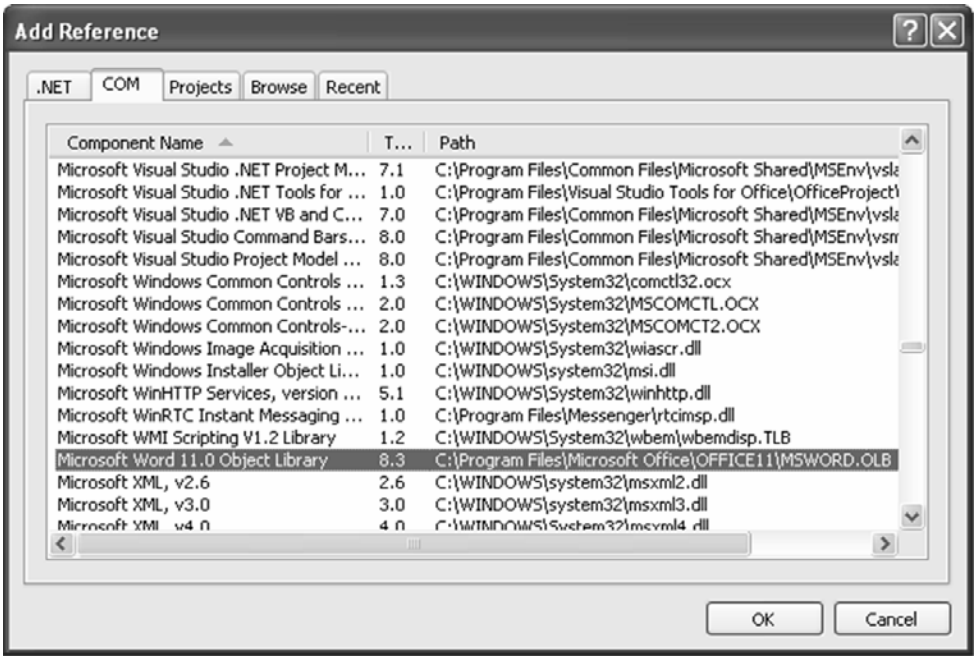


Figure 2-3: *Adding a reference to the Microsoft Word 2003 PIA.*

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.

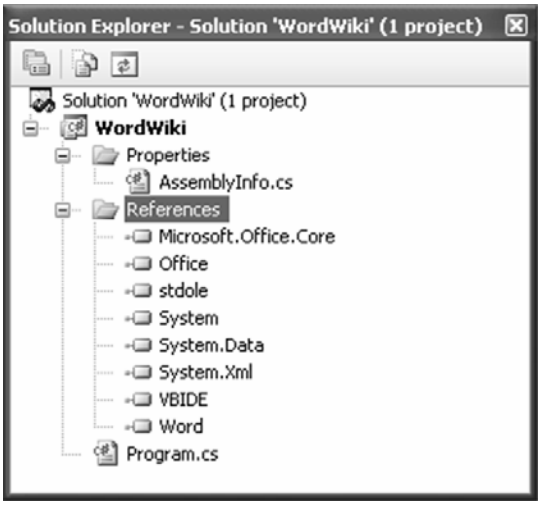


Figure 2-4: *When you add the Word 2003 PIA, dependent PIA references are automatically added to the project.*

Now that the proper references have been added to our console application, let's start writing code. Double click on 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 on the + sign to expand out the code where the using directives are placed. Add the following three using directives so we can use objects from the Word PIA and the Microsoft.Office.Core PIA as well as classes in the System.IO namespace.

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

We are now ready to write some real code that automates Word to create a table after reading a text input file in the wiki table format. The entire listing of our program is shown in Listing 2-3. Rather than explain every line of code in that listing, we will 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 will briefly touch on some objects in the Word object model here, but Chapters 6 through 8 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 Main method of our program class.

```
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:

```
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 wish to omit as shown here:

```
object missing = Type.Missing;
Word.Document theDocument = theApplication.Documents.Add(
      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. Once we know the number of columns and rows, we use the line of code below 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.

```
Word.Range range = theDocument.Range(ref missing, ref
missing);
```

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.

```
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 below. Note that as in most of the Office object models, the indices are 1-based meaning they start with one as the minimum value rather than being 0-based and starting with zero as the minimum value.

```
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 or 1) rather than a `bool`. This is one of many examples you will come across where the Office object models don't match .NET guidelines because of their origins in COM.

```
// Format table
table.Rows[1].Range.Bold = 1;
```

```
table.AutoFitBehavior(Word.WdAutoFitBehavior.wdAutoFitCon
tent);
```

Finally, there is some code at the end of the program that forces Word to quit without saving changes:

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

If we don't write this code, Word will stay running even after the console application exits. Once 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 will continue running. To force Word to exit we 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 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 file name 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 on the WordWiki project in Solution Explorer and choosing Properties. Then click on the Debug tab and set the Command line arguments field to the name of your text file.

Listing 2-3: The complete WordWiki implementation.

```
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;
      Word.Document theDocument =
theApplication.Documents.Add(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);
          columnCount = splitHeaderRow.Length - 2; //
Ignore the first & last separator
        }

        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)
      {
        string[] splitRow = r.Split(separators,
StringSplitOptions.None);

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

        rowIndex++;
      }

      // Format table
      table.Rows[1].Range.Bold = 1;

table.AutoFitBehavior(Word.WdAutoFitBehavior.wdAutoFitCon
tent);

      // Wait for input from the command line before
exiting
      System.Console.WriteLine("The table has been
generated.");
      System.Console.ReadLine();

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

Office Add-Ins

The second pattern used in Office development is the add-in pattern. This book will cover 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 2005 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 2005 code behind model for documents. Chapter 23 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 24 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.

There are some Office add-in technologies that this book will not discuss. Application level Smart Tags add-ins and Smart Documents add-ins are not discussed because VSTO provides a much easier way of accessing Smart Tag and 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 Tags and Smart Documents, see Chapter 15 and Chapter 16.

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 press the OK button.

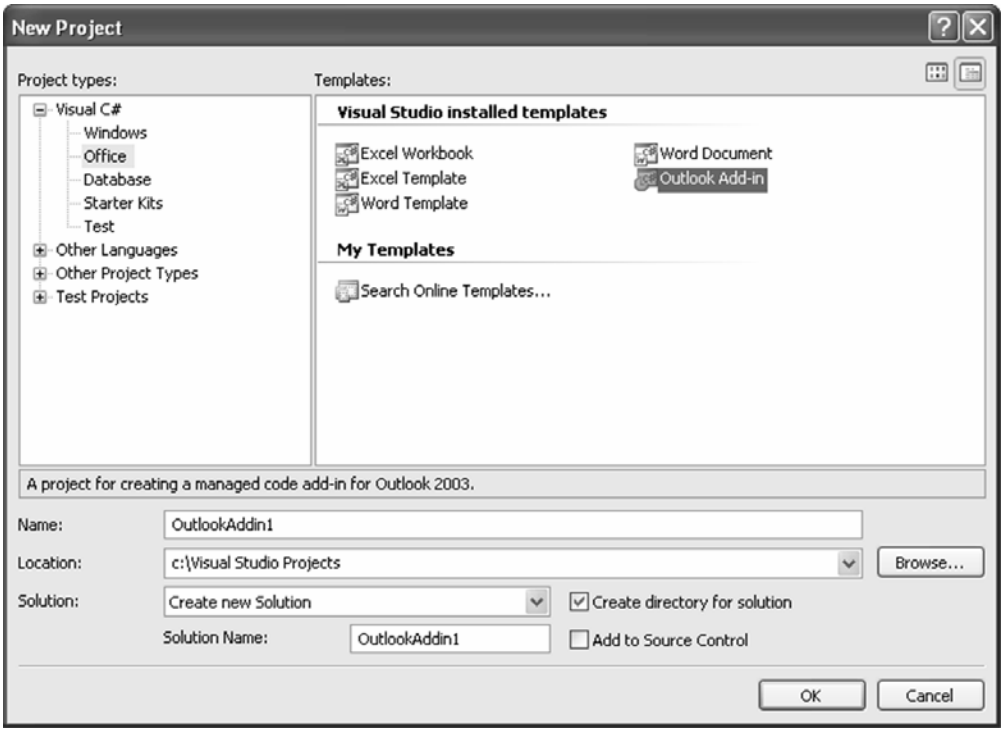


Figure 2-5: *Creating a new Outlook add-in project.*

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.

Figure 2-6: *The Outlook add-in project in Solution Explorer.*

If you double click on 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 like `this.Inspectors.Count` in the ThisApplication class.

Listing 2-4: The initial code in the ThisApplication class in an Outlook add-in project.

```
using System;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Outlook = Microsoft.Office.Interop.Outlook;

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

```
    private void ThisApplication_Shutdown(object sender,
System.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 may wonder about the use of "partial" in the class definition. VSTO uses partial classes which are a new feature of .NET that allows 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 partial class generated by VSTO that is hidden from you.

We are going to add to the code in Listing 2-4 to create an add-in to that will solve an annoying problem—people replying inadvertently to an e-mail sent out to a mailing 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 people 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 fires whenever a user sends an email. We will add additional code to the Startup method of the ThisApplication class to hookup an event handler for the ItemSend event as shown in Listing 2-5. Because the ThisApplication class derives from an aggregate of Outlook'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` instead of as a specific type. The ItemSend 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 if the Item parameter 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 if 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 they really want 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 ThisApplication Class in an Outlook add-in project that handles the ItemSend event and checks for more than 25 recipients.

```
using System;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Outlook = Microsoft.Office.Interop.Outlook;

namespace OutlookAddin1
{
  public partial class ThisApplication
  {
    private void ThisApplication_Startup(object sender,
System.EventArgs e)
    {
      this.ItemSend += new
Outlook.ApplicationEvents_11_ItemSendEventHandler(ThisApp
lication_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 they really want to send
this email
            string message = "Are you sure that you want
to send mail to " +
              recip.AddressEntry.Name + " " + " which
includes " +
              recip.AddressEntry.Members.Count + "
people?";

            string caption = "More than 25 recipients";
            MessageBoxButtons buttons =
MessageBoxButtons.YesNo;
            DialogResult result;

            result = MessageBox.Show(message, caption,
buttons);

            if (result == DialogResult.No)
            {
              Cancel = true;
              break;
            }
          }
        }
      }
    }

    private void ThisApplication_Shutdown(object sender,
System.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
```

```
  }
}
```

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.

VSTO Outlook add-ins are discussed in more detail in chapter 23. The Outlook object model is discussed in Chapters 9 through 11.

Code behind a Document

Visual Studio Tools for Office 2005 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. A VSTO project can have multiple startup classes that handle a Startup and Shutdown event raised on each startup class. In Word, there is only one startup class corresponding to the document. In Excel, there are multiple startup 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 startup classes created for you. VSTO provides context via the base class of the class you are writing code in. A VSTO class customizing an Excel worksheet derives from a base class that aggregates all the methods, properties, and events of an Excel worksheet. This enables you to write code like this in the Startup method of a worksheet class.

```
MessageBox.Show(this.Name + " is the sheet name.");
```

By using `this.Name`, you are referring to the Name property of the Excel Worksheet object aggregated by the base class. Listing 2-6 shows a VSTO 2005 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 as it is auto-generated by VSTO 2005 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 2005 code behind class.

```
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Excel = Microsoft.Office.Interop.Excel;
using Office = Microsoft.Office.Core;

namespace ExcelWorkbook1
{
  public partial class Sheet1
  {
    private void Sheet1_Startup(object sender,
System.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(this.Name + " is the sheet name.
");
    }

    private void Sheet1_Shutdown(object sender,
System.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 2005 Code behind a Document in Excel

In this section, we are going to create some simple code behind a document in Excel using VSTO 2005. First, start up VSTO 2005 and select the File / New / Project menu item. As we've seen previously, navigate to the Office node under the Visual C# root.

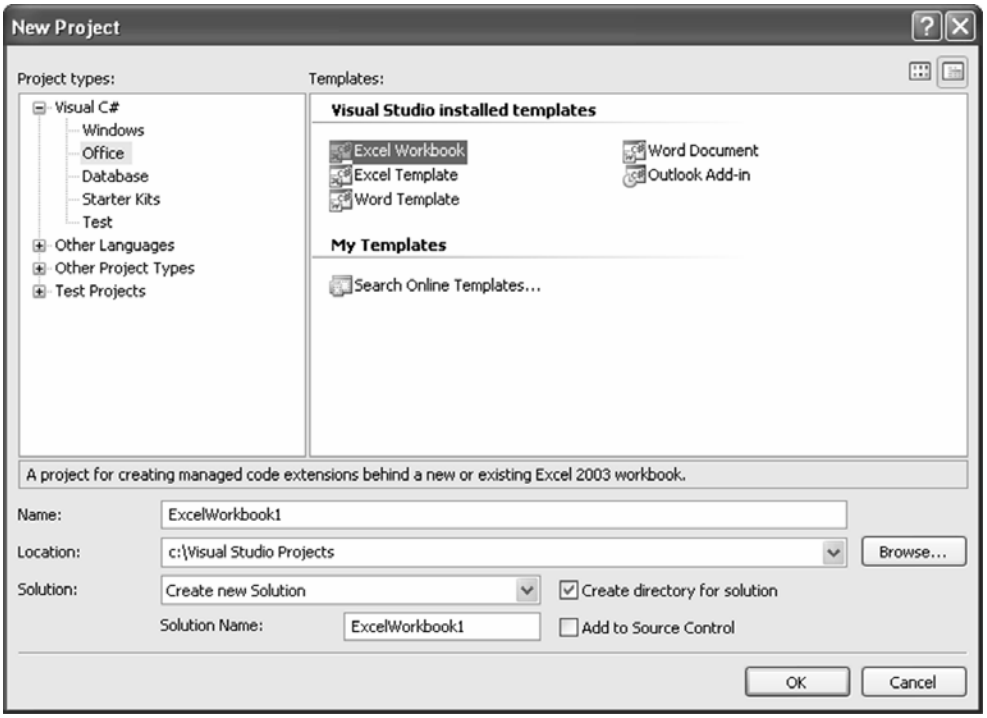


Figure 2-7: *Using the New Project dialog to create an Excel Workbook project.*

We will create an Excel Workbook project using C#. If you already have a workbook which you would like to add VSTO 2005 customization code behind, the dialog box shown in Figure 2-8 pops up and asks you where it can

be found. This time we will just start from scratch, creating a new, blank workbook.

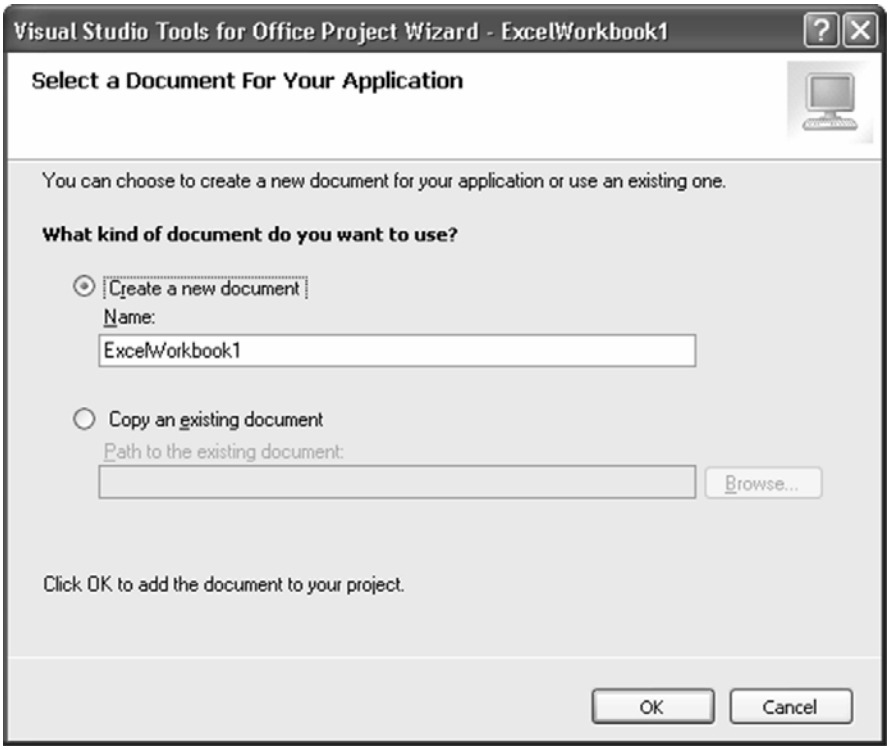


Figure 2-8: *Selecting the workbook to associate with your code behind.*

Once we have created the project, the designer view appears as shown in Figure 2-9.

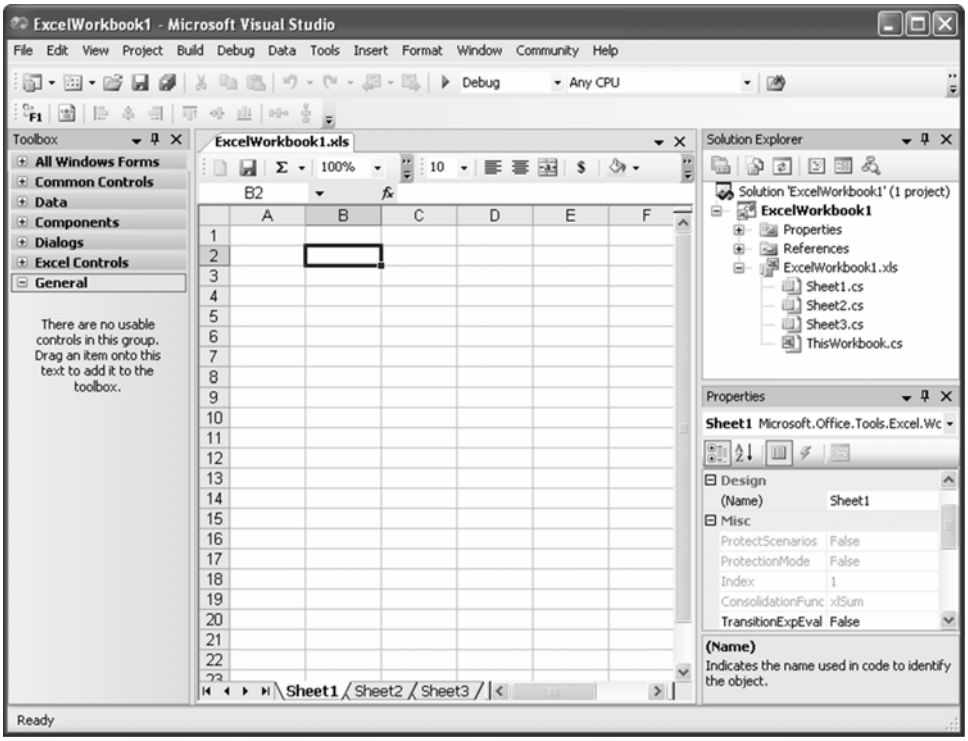


Figure 2-9: *The design view for VSTO 2005 Excel code behind.*

There are a few interesting things to notice in Figure 2-9. First, Excel is running inside the Visual Studio 2005 IDE as a designer, just the same as a Windows Forms designer would when developing a Windows Forms project.

Second, look at the menu bar as 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 sub menu to the Visual Studio menu such as Microsoft Office Excel Tools that can be selected to see the Excel Tools menu.

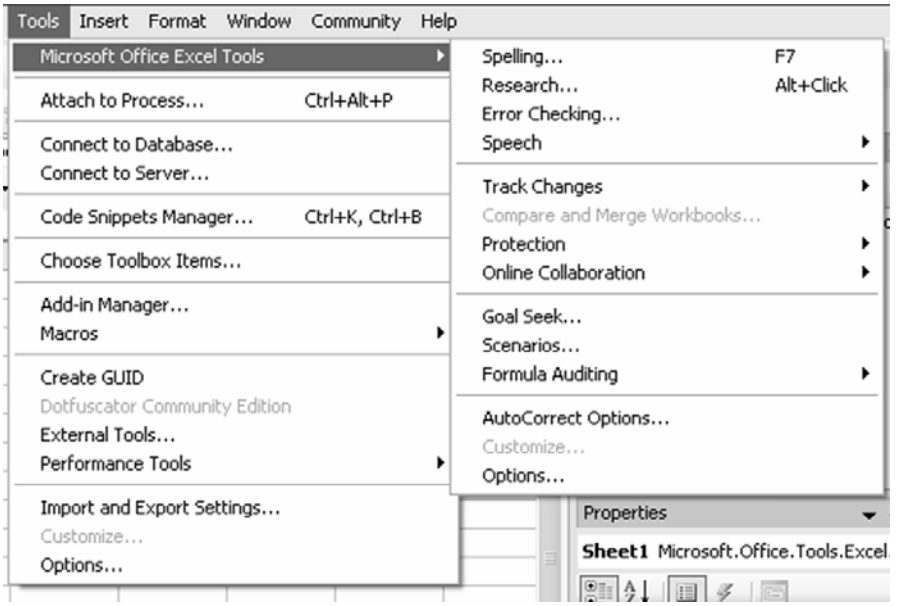


Figure 2-10: *The merging of Visual Studio and Excel menus.*

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 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, let's drag a ListObject from the toolbox onto the Sheet1 designer, and draw it to be ten rows by four columns as shown in Figure 2-11.

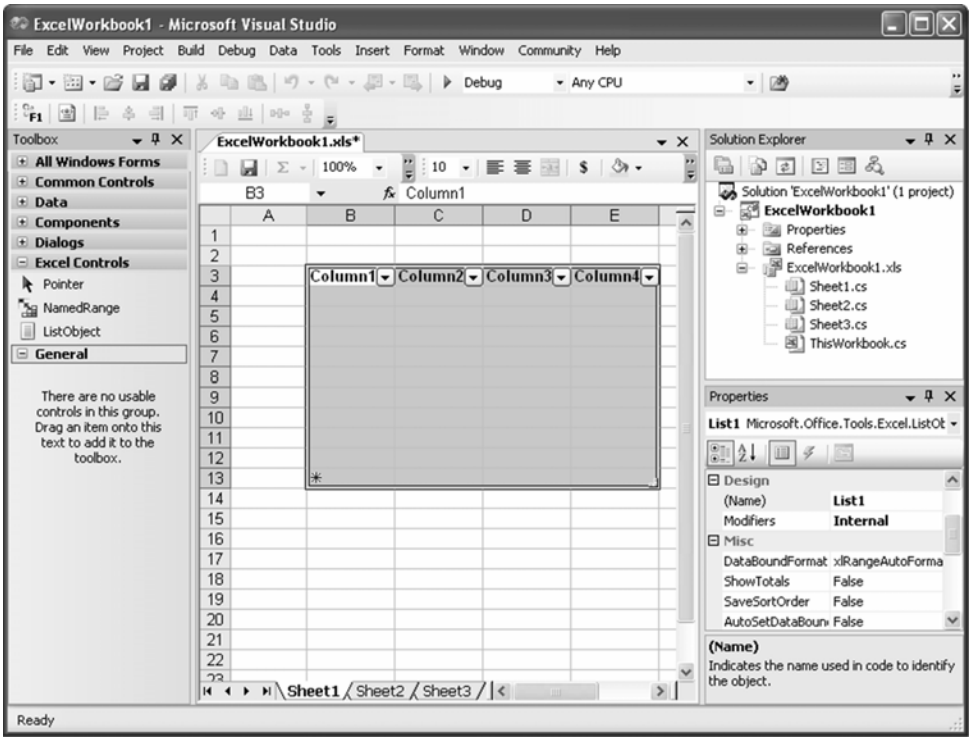


Figure 2-11: *Creating a ListObject in the designer.*

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 on Sheet1.cs in the Solution Explorer and select View Code. We are going to briefly illustrate two VSTO features—ActionsPane and ListObject databinding. 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. This 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 databind our list object to a randomly generated DataTable. This code is shown in Listing 2-7.

Listing 2-7: A VSTO 2005 code behind class that illustrates adding a control to the Document Actions task pane and databinding a ListObject control to a DataTable.

```
using System;
using System;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
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,
System.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,
System.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 ListObject as shown in Figure 2-12. Exit Excel to end your debugging session.

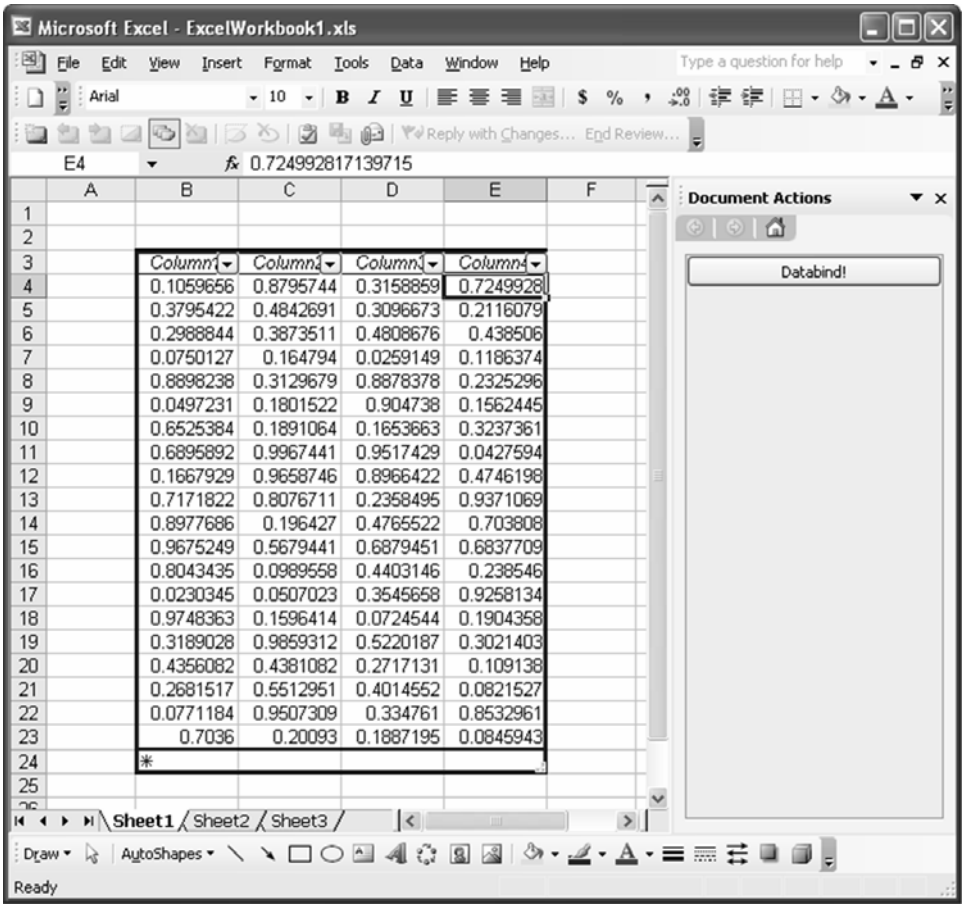


Figure 2-11: *The result of running Listing 2-7 and clicking on the button we added to the Document Actions task pane.*

We have briefly illustrated VSTO's support for the Document Actions task pane and the ability to databind that VSTO adds to Excel's ListObject. For more information on VSTO's support for the Document Actions task pane, see Chapter 15. For more information on VSTO's support for databinding, see Chapter 17.

Conclusion

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

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

Although we will not have more to say about automation executables, this chapter has only served as an introduction to add-ins and code behind documents. Chapter 23 covers VSTO add-ins for Outlook. Chapter 24 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 2005 in greater detail.

Microsoft .NET Development Series

John Montgomery, *Series Advisor*
Don Box, *Series Advisor*
Martin Heller, *Series Editor*

The **Microsoft .NET Development Series** is supported and developed by the leaders and experts of Microsoft development technologies including Microsoft architects and DevelopMentor instructors. The books in this series provide a core resource of information and understanding every developer needs in order to write effective applications and managed code. Learn from the leaders how to maximize your use of the .NET Framework and its programming languages.

Titles in the Series

Brad Abrams, *.NET Framework Standard Library Annotated Reference Volume 1: Base Class Library and Extended Numerics Library*, 0-321-15489-4

Brad Abrams and Tamara Abrams, *.NET Framework Standard Library Annotated Reference, Volume 2: Networking Library, Reflection Library, and XML Library*, 0-321-19445-4

Keith Ballinger, *.NET Web Services: Architecture and Implementation*, 0-321-11359-4

Bob Beauchemin, Niels Berglund, Dan Sullivan, *A First Look at SQL Server 2005 for Developers*, 0-321-18059-3

Don Box with Chris Sells, *Essential .NET, Volume 1: The Common Language Runtime*, 0-201-73411-7

Keith Brown, *The .NET Developer's Guide to Windows Security*, 0-321-22835-9

Mahesh Chand, *Graphics Programming with GDI+*, 0-321-16077-0

Anders Hejlsberg, Scott Wiltamuth, Peter Golde, *The C# Programming Language*, 0-321-15491-6

Alex Homer, Dave Sussman, Mark Fussell, *ADO.NET and System.Xml v. 2.0—The Beta Version*, 0-321-24712-4

Alex Homer, Dave Sussman, Rob Howard, *ASP.NET v. 2.0—The Beta Version*, 0-321-25727-8

James S. Miller and Susann Ragsdale, *The Common Language Infrastructure Annotated Standard*, 0-321-15493-2

Christian Nagel, *Enterprise Services with the .NET Framework: Developing Distributed Business Solutions with .NET Enterprise Services*, 0-321-24673-X

Fritz Onion, *Essential ASP.NET with Examples in C#*, 0-201-76040-1

Fritz Onion, *Essential ASP.NET with Examples in Visual Basic .NET*, 0-201-76039-8

Ted Pattison and Dr. Joe Hummel, *Building Applications and Components with Visual Basic .NET*, 0-201-73495-8

Dr. Neil Roodyn, *eXtreme .NET: Introducing eXtreme Programming Techniques to .NET Developers*, 0-321-30363-6

Chris Sells, *Windows Forms Programming in C#*, 0-321-11620-8

Chris Sells and Justin Gehtland, *Windows Forms Programming in Visual Basic .NET*, 0-321-12519-3

Paul Vick, *The Visual Basic .NET Programming Language*, 0-321-16951-4

Damien Watkins, Mark Hammond, Brad Abrams, *Programming in the .NET Environment*, 0-201-77018-0

Shawn Wildermuth, *Pragmatic ADO.NET: Data Access for the Internet World*, 0-201-74568-2

Paul Yao and David Durant, *.NET Compact Framework Programming with C#*, 0-321-17403-8

Paul Yao and David Durant, *.NET Compact Framework Programming with Visual Basic .NET*, 0-321-17404-6

For more information go to www.awprofessional.com/msdotnetseries/