49

Visual Studio Tools for Office

WHAT'S IN THIS CHAPTER?

- What types of projects you can create with VSTO and what capabilities you can include in these projects
- Fundamental techniques that apply to all types of VSTO solutions
- Using host items and host controls
- Building VSTO solutions with a custom UI

Visual Studio Tools for Office (VSTO) is a technology that enables you to customize and extend Microsoft Office applications and documents by using the .NET Framework. It also includes tools that you can use to make this customization easier in Visual Studio — for example, a visual designer for Office Ribbon controls.

VSTO is the latest in a long line of products that Microsoft has released to allow the customization of Office applications. The object model that you use to access Office applications has evolved over time. If you have used it in the past, then parts of it will be familiar to you. If you have programmed VBA add-ins for Office applications, then you will be well prepared for the techniques discussed in this chapter. However, the classes that VSTO makes available so that you can interact with Office have been extended beyond the Office object model. For example, the VSTO classes include .NET data binding functionality.

Up until Visual Studio 2008, VSTO was a separate download that you could obtain if you wanted to develop Office solutions. Starting from Visual Studio 2008, VSTO was integrated with the VS IDE. The previous version of VSTO, which was also known as VSTO 3, included full support for Office 2007 and many new features. This included the ability to interact with Word content controls, the visual ribbon designer mentioned previously, and more.

In Visual Studio 2010, VSTO 4 extends and improves upon the previous version. It makes deployment easier, and there is no longer a requirement to install Primary Interop Assemblies (PIAs) on client PCs — this is instead achieved through CLR 4 type embedding. Support for Office 2010 is also included.

This chapter does not assume any prior knowledge of VSTO or its predecessors.

VSTO OVERVIEW

VSTO consists of the following components:

➤ A selection of project templates that you can use to create various types of Office solutions
➤ Designer support for visual layout of ribbons, action panes, and custom task panels
➤ Classes built on top of the Office object model that provide extensive capabilities

VSTO supports 2003, 2007, and 2010 versions of Office. The VSTO class library comes in multiple flavors for each of these Office versions, which use different sets of assemblies. For simplicity, this chapter focuses on the 2007 version.

The general architecture of VSTO solutions is shown in Figure 49-1.

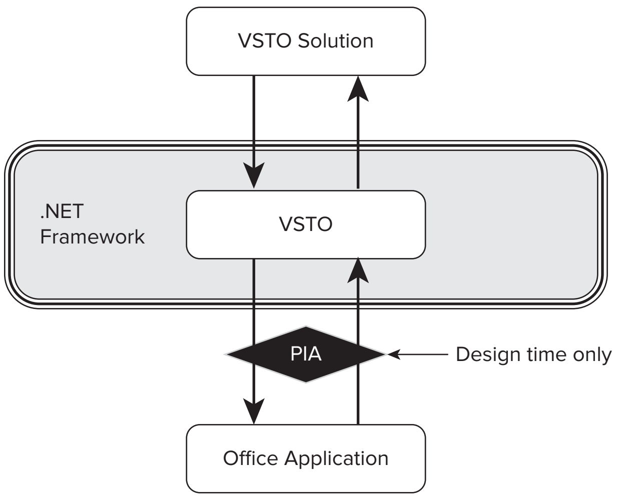
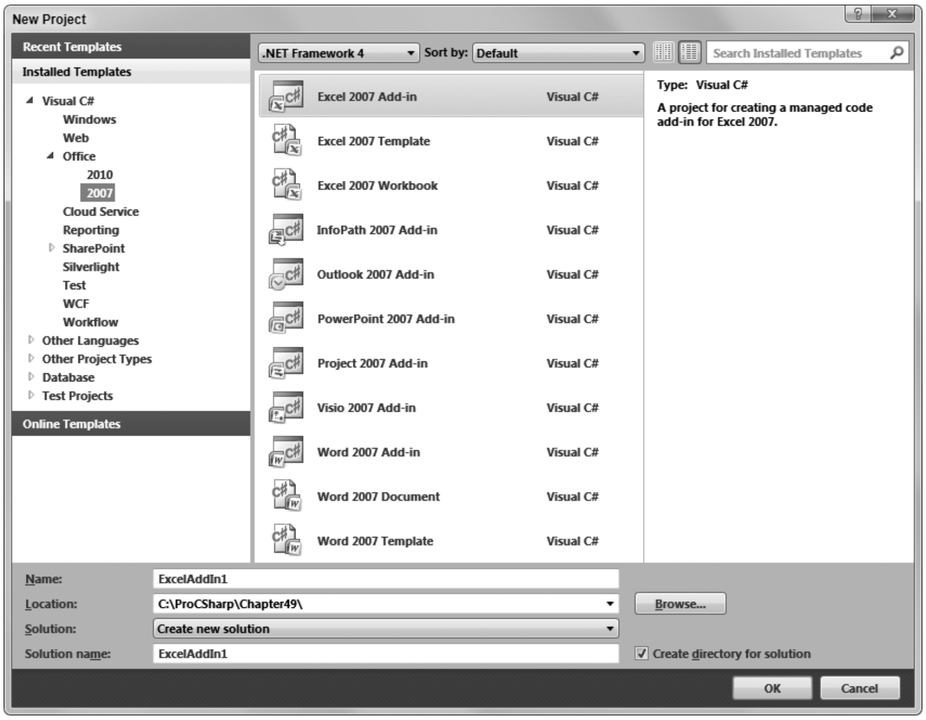


FIGURE 49-1

Note that the PIA shown in Figure 49-1 is only required at design time when targetting .NET 4, as type embedding is used when deployed to clients. This is not true for VSTO solutions that target .NET 3.5, which require PIAs on both the development and target machines.

Project Types

Figure 49-2 shows the project templates that are available in VS for Office 2007 (a similar list is available for Office 2010). In this chapter you'll concentrate on Office 2007.



Excel 2007 Add-in	Visual C#
Excel 2007 Template	Visual C#
Excel 2007 Workbook	Visual C#
InfoPath 2007 Add-in	Visual C#
Outlook 2007 Add-in	Visual C#
PowerPoint 2007 Add-in	Visual C#
Project 2007 Add-in	Visual C#
Visio 2007 Add-in	Visual C#
Word 2007 Add-in	Visual C#
Word 2007 Document	Visual C#
Word 2007 Template	Visual C#

Type: Visual C#

A project for creating a managed code add-in for Excel 2007.

Name: ExcelAddIn1

Location: C:\ProCSharp\Chapter49\

Solution: Create new solution

Solution name: ExcelAddIn1

☑ Create directory for solution

FIGURE 49-2

The VSTO project templates can be divided into the following categories:

- Document-level customizations
- Application-level add-ins
- SharePoint workflow templates (not shown in Figure 49-2 as these are in a separate template category)

Note that the previous version of VSTO included a fourth category, InfoPath form templates, which is no longer available in VSTO 4.

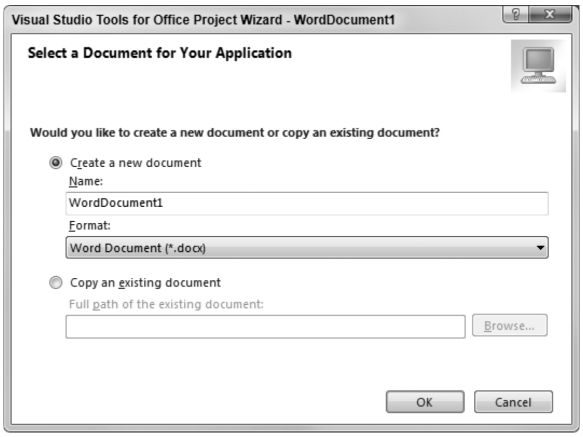
This chapter concentrates on the most commonly used project types, which are document-level customizations and application-level add-ins.

Document-Level Customizations

When you create a project of this type, you will generate an assembly that will be linked to an individual document — for example a Word document, Word template, or Excel workbook. When you load the document, the associated Office application will detect the customization, load the assembly, and make the VSTO customization available.

You might use a project of this type to provide additional functionality to a particular line-of-business document, or to a whole class of documents by adding customizations to a document template. You can include code that manipulates the document and the content of the document, including any embedded objects. You can also provide custom menus, including ribbon menus that you can create using the VS Ribbon Designer.

When you create a document-level project, you can choose to create a new document or to copy an existing document as a starting point for your development. You can also choose the type of document to create. For a Word document, for example, you can choose to create `.docx` (the default), `.doc`, or `.docm` documents (`.docm` is a macro-enabled document). The dialog box for this is shown in Figure 49-3.


Visual Studio Tools for Office Project Wizard - WordDocument1

Select a Document for Your Application

Would you like to create a new document or copy an existing document?

(●) Create a new document

Name: WordDocument1

Format: Word Document (*.docx)

( ) Copy an existing document

Full path of the existing document:

FIGURE 49-3

Application-Level Add-Ins

Application-level add-ins are different from document-level customizations in that they are available throughout their targeted Office application. You can access add-in code, which might include menus, document manipulations, and so on, regardless of what documents are loaded.

When you start an Office application such as Word, it will look for associated add-ins that have entries in the registry and will load any assemblies that it needs to.

SharePoint Workflow Templates

These projects provide a template to create SharePoint workflow applications. These are used to manage the flow of documents within SharePoint processes. By creating a project of this type, you can execute custom code at key times during the document lifecycle.

Project Features

There are several features that you can use in the various VSTO project types, such as interactive panes and controls. The project type you use determines the features that are available to you. The following tables list these features according to the projects in which they are available.

Document-Level Customization Features

FEATURE	DESCRIPTION
Actions pane	Actions panes are dialog boxes that are hosted inside the action pane of Word or Excel. You can display whatever controls you like here, which makes this an extremely versatile way of extending documents and applications.
Data cache	Data caching enables you to store data that is used in your documents externally to those documents in cached data islands. These data islands can be updated from data sources or manually, and enable the Office documents to access data when data sources are offline or unavailable.
Endpoints for VBA code	As discussed earlier, VSTO enables VBA interoperability. In document-level customizations, you can provide endpoint methods that can be called from VBA code.
Host controls	Host controls are extended wrappers around existing controls in the Office object model. You can manipulate and data-bind to these objects.
Smart Tags	Smart Tags are objects that are embedded in Office documents and that have typed content. They are automatically detected within the content of Office documents; for example, a stock quote Smart Tag is added automatically when the application detects appropriate text. You can create your own Smart Tag types and define operations that can be performed on them.

FEATURE	DESCRIPTION
Visual document designer	When you work with document customization projects, the Office object model is used to create a visual design surface that you can use to lay out controls interactively. The toolbars and menus shown in the designer are, as you will see later in this chapter, fully functional.

Application-Level Add-In Features

FEATURE	DESCRIPTION
Custom task pane	Task panes are typically docked to one edge of an Office application and provide a variety of functionality. For example, Word has a task pane used for manipulating styles. As with action panes, these give you a great deal of flexibility.
Cross-application communication	Once you have created an add-in for one Office application, you can expose that functionality to other add-ins. You could, for example, create a financial calculating service in Excel and then use that service from Word — without creating a separate add-in.
Outlook form regions	You can create form regions that can be used in Outlook.

Features Usable In All Project Types

FEATURE	DESCRIPTION
ClickOnce deployment	You can distribute any VSTO project that you create to end users through ClickOnce deployment methods. This enables users to stay up-to-date with updates to your document- and application-level solutions by detecting changes to the application manifest.
Ribbon menus	Ribbon menus are used in all Office applications, and VSTO includes two ways to create your own ribbon menus. You can either use XML to define a ribbon or use the Ribbon Designer. Typically, you will use the Ribbon Designer as it is much easier to use, although you may want the XML version for backwards compatibility.

VSTO PROJECT FUNDAMENTALS

Now that you have seen what is included in VSTO, it is time to look at the more practical side of things, and how you can build VSTO projects. The techniques demonstrated in this section are general ones that apply to all types of VSTO projects.

In this section, you will look at the following:

- Office object model
- VSTO namespaces
- Host items and host controls
- Basic VSTO project structure
- The Globals class
- Event handling

Office Object Model

The Office 2007 and 2010 suites of applications expose their functionality through a COM object model. You can use this object model directly from VBA to control just about any aspect of Office functionality. The Office object model was introduced in Office 97, and has evolved since then as functionality in Office has changed.

There are a huge number of classes in the Office object model, some of which are used across the suite of Office applications and some of which are specific to individual applications. For example, the Word 2007 object model includes a `Documents` collection representing the currently loaded objects, each of which is represented by a `Document` object. In VBA code, you can access documents by name or index and call methods to perform operations on them. For example, the following VBA code closes the document with the name `My Document` without saving changes:

```
Documents("My Document").Close SaveChanges:= wdDoNotSaveChanges
```

The Office object model includes named constants (such as `wdDoNotSaveChanges` in the preceding code) and enumerations to make it easier to use.

VSTO Namespaces

VSTO contains a collection of namespaces, which contain types that you can use to program against the Office object model. Many of the types in these namespaces map directly to types in the Office object model. These are accessed through Office PIAs at design time, and through embedded type information when solutions are deployed. Because of this type embedding, you will mostly use interfaces to access the Office object model. VSTO also contains types that do not map directly, or are unrelated to the Office object model. For example, there are a lot of classes that are used for designer support in VS.

The types that do wrap or communicate with objects in the Office object model are divided into namespaces. The namespaces that you will use for Office development are summarized in the following table.

NAMESPACE	DESCRIPTION
`Microsoft.Office.Core`, `Microsoft.Office.Interop.*`	These namespaces contain interfaces and thin wrappers around the office object model and, so, provide the base functionality for working with the Office classes. There are several nested namespaces in the `Microsoft.Office.Interop` namespace for each of the Office products.
`Microsoft.Office.Tools`	This namespace contains general types that provide VSTO functionality and base classes for many of the classes in nested namespaces. For example, this namespace includes the classes required to implement action panes in document-level customizations and the base class for application-level add-ins.
`Microsoft.Office.Tools.Excel`, `Microsoft.Office.Tools.Excel.*`	These namespaces contain the types required to interact with the Excel application and Excel documents.
`Microsoft.Office.Tools.Outlook`	These namespaces contain the types required to interact with the Outlook application.
`Microsoft.Office.Tools.Ribbon`	This namespace includes the types required to work with and create your own ribbon menus.
`Microsoft.Office.Tools.Word`, `Microsoft.Office.Tools.Word.*`	These namespaces contain the types required to interact with the Word application and Word documents.
`Microsoft.VisualStudio.Tools.*`	These namespaces provide the VSTO infrastructure that you work with when you develop VSTO solutions in VS.

Host Items and Host Controls

Host items and host controls are interfaces that have been extended to make it easier for document-level customizations to interact with Office documents. These interfaces simplify your code as they expose .NET-style events and are fully managed. The “host” part of the name of host items and host controls references the fact that these interfaces wrap and extend the native Office objects.

Often when you use host items and host controls, you will find that it is necessary to use the underlying interop types as well. For example, if you create a new Word document, then you receive a reference to the interop Word document type rather than the Word document host item. You need to be aware of this and write your code accordingly.

There are host items and host controls for both Word and Excel document-level customizations.

Word

There is a single host item for Word, `Microsoft.Office.Tools.Word.Document`. This represents a Word document. As you might expect, this interface has an enormous number of methods and properties that you can use to interact with Word documents.

There are 12 host controls for Word, as shown in the following table, all of which are found in the `Microsoft.Office.Tools.Word` namespace.

CONTROL	DESCRIPTION
`Bookmark`	This control represents a location within the Word document. This might be a single location or a range of characters.
`XMLNode, XmlNodes`	These controls are used when the document has an attached XML schema. They allow you to reference document content by the XML node location of that content. You can also manipulate the XML structure of a document with these controls.
`ContentControl`	This interface shares a base interface (`ContentControlBase`) with the remaining eight controls in this table, and enables you to deal with Word content controls. A content control is a control that presents content as a control or that enables functionality above and beyond that offered by plain text in a document.
`BuildingBlockGalleryContentControl`	This control enables you to add and manipulate document building blocks, such as formatted tables, cover pages, and so on.
`ComboBoxContentControl`	This control represents content formatted as a combo box.
`DatePickerContentControl`	This control represents content formatted in a date picker.
`DropDownListContentControl`	This control represents content formatted as a drop-down list.
`GroupContentControl`	This control represents content that is a grouped collection of other content items, including text and other content controls.
`PictureContentControl`	This control represents an image.
`RichTextContentControl`	This control represents a block of rich text content.
`PlainTextContentControl`	This control represents a block of plain text content.

Excel

There are three host items and four host controls for Excel, all of which are contained in the `Microsoft.Office.Tools.Excel` namespace.

The Excel host items are shown in the following table.

HOST ITEM	DESCRIPTION
`Workbook`	This host item represents an entire Excel workbook, which may contain multiple worksheets and chartsheets.
`Worksheet`	This host item is used for individual worksheets within a workbook.
`Chartsheet`	This host item is used for individual chartsheets within a workbook.

The Excel host controls are shown in the following table.

CONTROL	DESCRIPTION
`Chart`	This control represents a chart that is embedded in a worksheet.
`ListObject`	This control represents a list in a worksheet.
`NamedRange`	This control represents a named range in a worksheet.
`XmlMappedRange`	This control is used when an Excel spreadsheet has an attached schema, and is used to manipulate ranges that are mapped to XML schema elements.

Basic VSTO Project Structure

When you first create a VSTO project, the files you start with vary according to the project type, but there are some common features. In this section, you will see what constitutes a VSTO project.

Document-Level Customization Project Structure

When you create a document-level customization project, you will see an entry in Solution Explorer that represents the document type. This may be:

- A `.docx` file for a Word document
- A `.dotx` file for a Word template
- A `.xlsx` file for an Excel workbook
- A `.xltx` file for an Excel template

Each of these has a designer view and a code file, which you will see if you expand the item in Solution Explorer. The Excel templates also include sub-items representing the workbook as a whole and each spreadsheet in the workbook. This structure enables you to provide custom functionality on a per-sheet or per-workbook basis.

If you view the hidden files in one of these projects, you will see several designer files that you can look at to see the template-generated code. Each Office document item has an associated class from the VSTO namespaces, and the classes in the code files derive from these classes. These classes are defined as partial class definitions so that your custom code is separated from the code generated by the visual designer, similar to the structure of Windows Forms applications.

For example, the Word document template provides a class that derives from the `Microsoft.Office.Tools.Word.DocumentBase`. This class exposes the `Document` host item through a `Base` property, and is contained in `ThisDocument.cs`, as follows:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Xml.Linq;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Office = Microsoft.Office.Core;
using Word = Microsoft.Office.Interop.Word;

namespace WordDocument1
{
    public partial class ThisDocument
    {
        private void ThisDocument_Startup(object sender, System.EventArgs e)
        {
        }
```

```
            private void ThisDocument_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(ThisDocument_Startup);
                this.Shutdown += new System.EventHandler(ThisDocument_Shutdown);
            }
            #endregion
        }
    }
```

This template-generated code includes aliases for the two main namespaces that you will use when creating a document-level customization for Word, `Microsoft.Office.Core` for the main VSTO Office classes and `Microsoft.Office.Interop.Word` for Word-specific classes. Note that if you want to use Word host controls, then you would also add a `using` statement for the `Microsoft.Office.Tools.Word` namespace. The template-generated code also defines two event handler hooks that you can use to execute code when the document is loaded or unloaded, `ThisDocument_Startup()` and `ThisDocument_Shutdown()`.

Every one of the document-level customization project types has a similar structure in its code file (or, in the case of Excel, code files). There are namespace aliases defined for you and handlers for the various `Startup` and `Shutdown` events that the VSTO classes define. From this starting point, you add dialog boxes, action panes, ribbon controls, event handlers, and custom code to define the behavior of your customization.

With document-level customizations, you can also customize the document or documents through the document designer. Depending on the type of solution you are creating, this might involve adding boilerplate content to templates, interactive content to documents, or something else. The designers are effectively hosted versions of Office applications, and you can use them to enter content just as you can in the applications themselves. However, you can also add controls such as host controls and Windows Forms controls to documents, and code around these controls.

Application-Level Add-In Project Structure

When you create an application-level add-in, there will be no document or documents in Solution Explorer. Instead, you will see an item representing the application that you are creating an add-in for, and if you expand this item, you will see a file called `ThisAddIn.cs`. This file contains a partial class definition for a class called `ThisAddIn`, which provides the entry point for your add-in. This class derives from `Microsoft.Office.Tools.AddInBase`, which provides code add-in functionality.

For example, the code generated by the Word add-in template is as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;
using Microsoft.Office.Tools.Word;
```

```
namespace WordAddIn1
{
    public partial class ThisAddIn
    {
        private void ThisAddIn_Startup(object sender, System.EventArgs e)
        {
        }
        private void ThisAddIn_Shutdown(object sender, System.EventArgs e)
        {
        }
        #region VSTO 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(ThisAddIn_Startup);
            this.Shutdown += new System.EventHandler(ThisAddIn_Shutdown);
        }
        #endregion
    }
}
```

As you can see, this structure is very similar to the structure used in document-level customizations. It includes aliases for the same `Microsoft.Office.Core` and `Microsoft.Office.Interop.Word` namespaces, and gives you event handlers for `Startup` and `Shutdown` events (`ThisAddIn_Startup()` and `ThisAddIn_Shutdown()`). These events are slightly different from the document ones, as they are raised when the add-in is loaded or unloaded rather than when individual documents are opened or closed.

You proceed to customize application-level add-ins much as you do document-level customizations: by adding ribbon controls, task panes, and additional code.

The Globals Class

All VSTO project types define a class called `Globals` that gives you global access to the following:

- For document-level customizations, all documents in the solution. These are exposed through members with names that match the document class names — for example, `Globals.ThisWorkbook` and `Globals.Sheet1`.
- For application-level add-ins, the add-in object. This is exposed through `Globals.ThisAddIn`.
- For Outlook add-in projects, all Outlook form regions.
- All ribbons in the solution, through the `Globals.Ribbons` property.
- An interface that derives from `Microsoft.Office.Tools.Factory` that exposes project-type-specific utility methods, exposed through the `Factory` property.

Behind the scenes, the `Globals` class is created through a series of partial definitions in the various designer-maintained code files in your solution. For example, the default `Sheet1` worksheet in an Excel Workbook project includes the following designer-generated code:

```
internal sealed partial class Globals
{
   private static Sheet1 _Sheet1;
   internal static Sheet1 Sheet1
   {
      get
      {
         return _Sheet1;
      }
```

```
            set
            {
               if ((_Sheet1 == null))
               {
                  _Sheet1 = value;
               }
               else
               {
                  throw new System.NotSupportedException();
               }
            }
         }
      }
```

This code adds the `Sheet1` member to the `Globals` class.

The `Globals` class also allows you to transform an interop type to a VSTO type by using the `Globals.Factory.GetVstoObject()` method. This will obtain a VSTO `Workbook` interface from a `Microsoft.Office.Interop.Excel.Workbook` interface, for example. There is also a `Globals.Factory.HasVstoObject()` method that you can use to determine if such a transformation is possible.

Event Handling

Earlier in this chapter, you saw how the host item and host control classes expose events that you can handle. Unfortunately, this is not the case for the interop classes. There are a few events that you can use, but for the most part, you will find it difficult to create event-driven solutions by using these events. Most often, to respond to events you should focus on the events exposed by host items and host controls.

The obvious problem here is that there are no host items or host controls for application-level add-in projects. Sadly, this is a problem that you must learn to live with when you use VSTO. However, the most common events that you are likely to listen for in add-ins are those associated with ribbon menu and task pane interaction. You design ribbons with the integrated ribbon designer, and you can respond to any events generated by the ribbon to make the control interactive. Task panes are usually implemented as Windows Forms user controls (although you can use WPF), and you can use Windows Forms events here. This means that you will not often encounter situations in which there is no event available for the functionality you require.

When you do need to use an Office interop-exposed event, you will find that events are exposed through interfaces on these objects. Consider a Word Add-In project. The `ThisAddIn` class in this project exposes a property called `Application` through which you can obtain a reference to the Office application. This property is of type `Microsoft.Office.Interop.Word.Application`, and exposes events through the `Microsoft.Office.Interop.Word.ApplicationEvents4_Event` interface. This interface exposes a total of 29 events (which really doesn't seem to be a lot for an application as complex as Word, does it?). You can handle, for example, the `DocumentBeforeClose` event to respond to Word document close requests.

BUILDING VSTO SOLUTIONS

The previous sections explained what VSTO projects are, how they are structured, and the features that you can use in the various project types. In this section, you look at implementing VSTO solutions.

Figure 49-4 outlines the structure of document-level customization solutions.

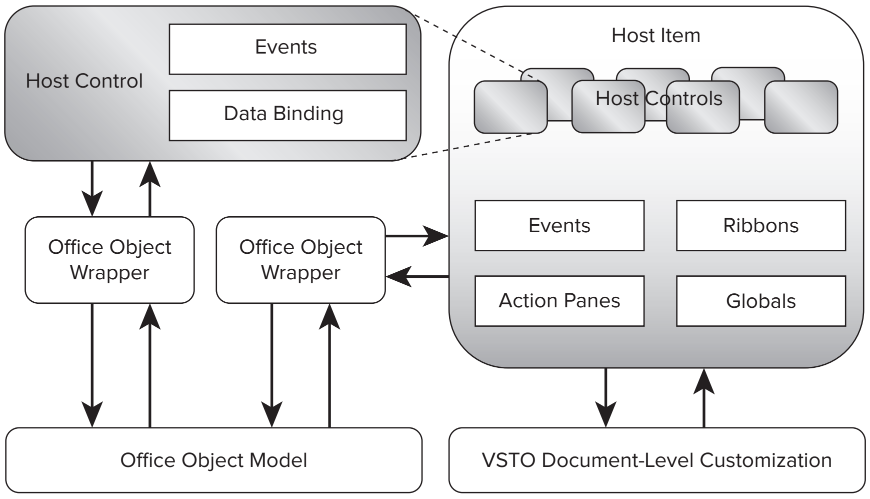


FIGURE 49-4

For document-level customizations you will interact with at least one host item, which will typically contain multiple host controls. You may use Office object wrappers directly, but for the most part, you will access the Office object model and its functionality through host items and host controls.

You will make use of host item and host control events, data binding, ribbon menus, action panes, and global objects in your code.

Figure 49-5 outlines the structure of application-level add-in solutions.

In this slightly simpler model, you are more likely to use the thinner wrappers around Office objects directly, or at least through the add-in class that encapsulates your solution. You will also use events exposed by the add-in class, ribbon menus, task panes, and global objects in your code.

In this section, you will look at both of these types of applications as appropriate, as well as the following topics:

- ➤ Managing application-level add-ins
- ➤ Interacting with applications and documents
- ➤ UI customization

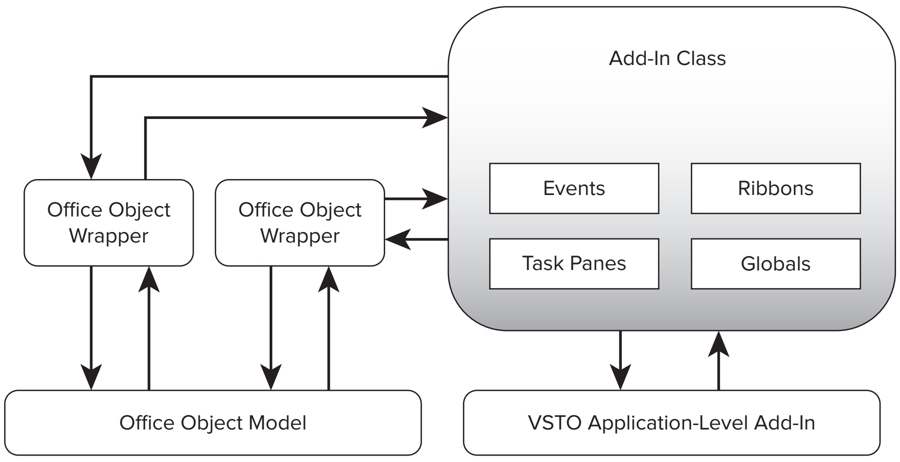
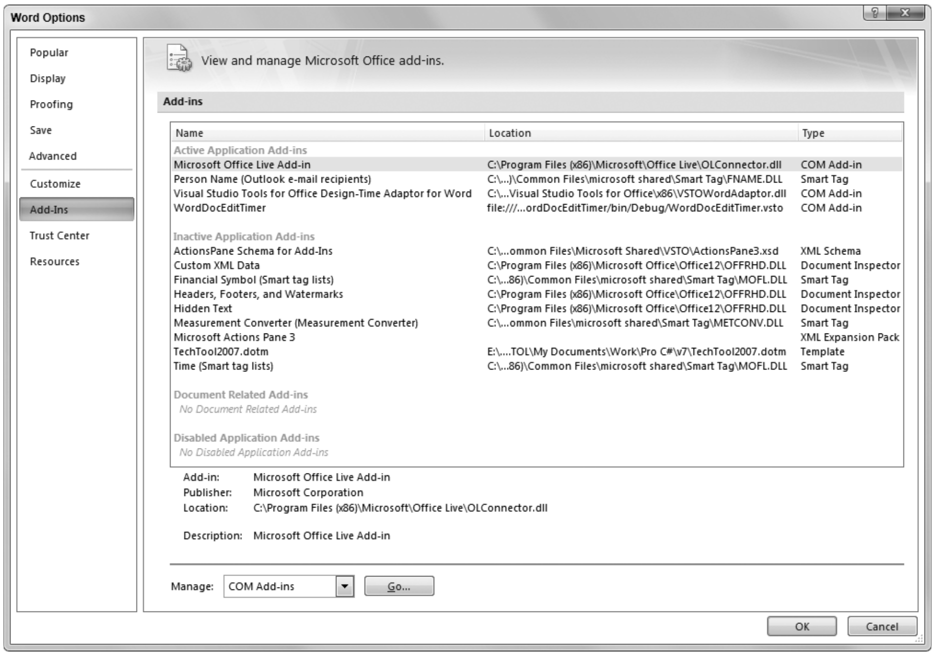


FIGURE 49-5

Managing Application-Level Add-Ins

One of the first things you will find when you create an application-level add-in is that VS carries out all the steps necessary to register the add-in with the Office application. This means that registry entries are added so that when the Office application starts, it will automatically locate and load your assembly. If you subsequently want to add or remove add-ins, then you must either navigate through Office application settings or manipulate the registry manually.

For example, in Word, you must open the Office Button menu, click Word Options, and select the Add-Ins tab, as shown in Figure 49-6.



View and manage Microsoft Office add-ins.

Add-ins

Name	Location	Type
Active Application Add-ins		
Microsoft Office Live Add-in	C:\Program Files (x86)\Microsoft\Office Live\OLConnector.dll	COM Add-in
Person Name (Outlook e-mail recipients)	C:\...)\Common Files\microsoft shared\Smart Tag\FNAME.DLL	Smart Tag
Visual Studio Tools for Office Design-Time Adaptor for Word	C:\...Visual Studio Tools for Office\x86\VSTOWordAdaptor.dll	COM Add-in
WordDocEditTimer	file:///...ordDocEditTimer/bin/Debug/WordDocEditTimer.vsto	COM Add-in
Inactive Application Add-ins		
ActionsPane Schema for Add-Ins	C:\...ommon Files\Microsoft Shared\VSTO\ActionsPane3.xsd	XML Schema
Custom XML Data	C:\Program Files (x86)\Microsoft Office\Office12\OFFRHD.DLL	Document Inspector
Financial Symbol (Smart tag lists)	C:\...86)\Common Files\microsoft shared\Smart Tag\MOFL.DLL	Smart Tag
Headers, Footers, and Watermarks	C:\Program Files (x86)\Microsoft Office\Office12\OFFRHD.DLL	Document Inspector
Hidden Text	C:\Program Files (x86)\Microsoft Office\Office12\OFFRHD.DLL	Document Inspector
Measurement Converter (Measurement Converter)	C:\...ommon Files\microsoft shared\Smart Tag\METCONV.DLL	Smart Tag
Microsoft Actions Pane 3		XML Expansion Pack
TechTool2007.dotm	E:\....TOL\My Documents\Work\Pro C#\v7\TechTool2007.dotm	Template
Time (Smart tag lists)	C:\...86)\Common Files\microsoft shared\Smart Tag\MOFL.DLL	Smart Tag
Document Related Add-ins		
No Document Related Add-ins		
Disabled Application Add-ins		
No Disabled Application Add-ins		

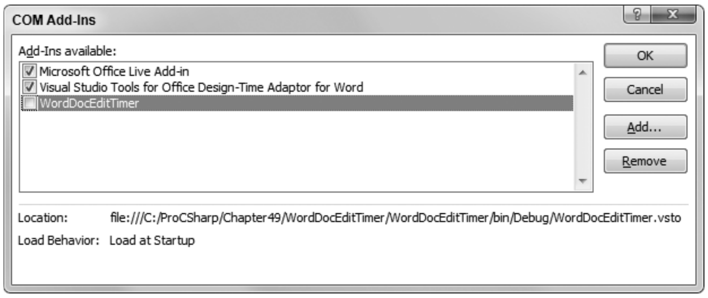
Add-in: Microsoft Office Live Add-in
Publisher: Microsoft Corporation
Location: C:\Program Files (x86)\Microsoft\Office Live\OLConnector.dll

Description: Microsoft Office Live Add-in

Manage: COM Add-ins

FIGURE 49-6

Figure 49-6 shows an add-in that has been created with VSTO: `WordDocEditTimer`. To add or remove add-ins, you must select COM Add-Ins in the Manage drop-down (the default option) and click the Go button. The dialog box that appears is shown in Figure 49-7.



Add-Ins available:
- [✓] Microsoft Office Live Add-in
- [✓] Visual Studio Tools for Office Design-Time Adaptor for Word
- [ ] WordDocEditTimer

Location: file:///C:/ProCSharp/Chapter49/WordDocEditTimer/WordDocEditTimer/bin/Debug/WordDocEditTimer.vsto
Load Behavior: Load at Startup

FIGURE 49-7

You can unload add-ins by deselecting them in the COM Add-Ins dialog box (as shown in Figure 49-7), and you can add new add-ins or remove old ones with the Add and Remove buttons.

Interacting with Applications and Documents

Whatever type of application you are creating, you will want to interact with the host application and/or documents in the host operation. In part, this includes using UI customizations, which you learn about in the next section. However, you may also need to monitor documents within an application, which means that you must handle some Office object model events. For example, to monitor documents in Word, you require event handlers for the following events of the `Microsoft.Office.Interop.Word .ApplicationEvents4_Event` interface:

- `DocumentOpen` — Raised when a document is opened.
- `NewDocument` — Raised when a new document is created.
- `DocumentBeforeClose` — Raised when a document is saved.

Also, when Word first starts, it will have a document loaded, which will either be a blank new document or a document that was loaded.

The downloadable code for this chapter includes an example called `WordDocEditTimer`*, which maintains a list of edit times for Word documents. Part of the functionality of this application is to monitor the documents that are loaded, for reasons that are explained later. Because this example also uses a custom task pane and ribbon menu, you will look at it after covering those topics.*

You can access the currently active document in Word through `ThisAddIn.Application.ActiveDocument` property, and the collection of open documents through `ThisAddIn.Application.Documents`. Similar properties exist for the other Office applications with a Multiple Document Interface (MDI). You can manipulate various properties of documents through the properties exposed by, for example, the `Microsoft.Office.Interop.Word.Document` class.

One point to note here is that the number of classes and class members you must deal with when developing VSTO solutions is, frankly, enormous. Until you get used to it, it can be difficult to locate the features you are after. For example, it is not obvious why in Word the current active selection is available not through the active document, but through the application (through the `ThisAddIn.Application.Selection` property).

The selection is useful for inserting, reading, or replacing text through the `Range` property. For example:

```
ThisAddIn.Application.Selection.Range.Text = "Inserted text";
```

Unfortunately, there is not enough space in this chapter to cover the object libraries in great depth. Instead, you will learn about the object libraries as they are relevant to the ongoing discussion.

UI Customization

Perhaps the most important aspect of the latest release of VSTO is the flexibility that is available for customizing the UI of your customizations and add-ins. You can add content to any of the existing ribbon menus, add completely new ribbon menus, customize task panes by adding action panes, add completely new task panes, and integrate Windows Forms and WPF forms and controls.

In this section, we look at each of these subjects.

Ribbon Menus

You can add ribbon menus to any of the VSTO projects that you are looking at in this chapter. When you add a ribbon, you will see the designer window shown in Figure 49-8.

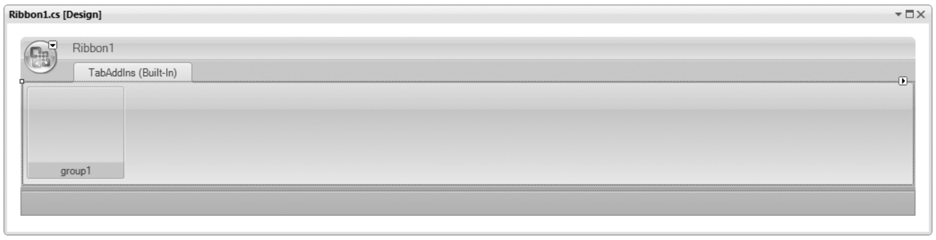


FIGURE 49-8

The designer allows you to customize this ribbon by adding controls to the Office button menu (shown in the top left of Figure 49-8) and to groups on the ribbon. You can also add additional groups.

The classes used in ribbons are found in the `Microsoft.Office.Tools.Ribbon` namespace. This includes the ribbon class that you derive from to create a ribbon, `RibbonBase`. This class can contain `RibbonTab` objects, each of which includes content for a single tab. Tabs contain `RibbonGroup` objects, like the `group1` group in Figure 49-8. These tabs can contain a variety of controls.

It is possible for the groups on a tab to be positioned on a completely new tab, or on one of the existing tabs in the Office application that you are targeting. Where the groups appear is determined by the `RibbonTab.ControlId` property. This property has a `ControlIdType` property, which you can set to `RibbonControlIdType.Custom` or `RibbonControlIdType.Office`. If you use `Custom`, then you must also set `RibbonTab.ControlId.CustomId` to a `string` value, which is the tab identifier. You can use any identifier you like here. However, if you use `Office` for `ControlIdType`, then you must set `RibbonTab.ControlId.OfficeId` to a `string` value that matches one of the identifiers used in the Office product you are using. For example, in Excel you could set this property to `TabHome` to add groups to the Home tab, `TabInsert` for the Insert tab, and so on. The default for add-ins is `TabAddIns`, which will be shared by all add-ins.

Many tabs are available, especially in Outlook; you can download a series of spreadsheets containing the full list from: `www.microsoft.com/downloads/details.aspx?FamilyID=4329D9E9-4D11-46A5-898D-23E4F331E9AE&displaylang=en`.

Once you have decided where to put your ribbon groups, you can add any of the controls shown in the following table.

CONTROL	DESCRIPTION
`RibbonBox`	This is a container control that you can use to lay out other controls in a group. You can lay out controls in a `RibbonBox` horizontally or vertically by changing the `BoxStyle` property to `RibbonBoxStyle.Horizontal` or `RibbonBoxStyle.Vertical`.
`RibbonButton`	You can use this control to add a small or large button with or without a text label to a group. Set the `ControlSize` property to `RibbonControlSize.RibbonControlSizeLarge` or `RibbonControlSize.RibbonControlSizeRegular` to control the size. The button has a `Click` event handler that you can use to respond to interaction. You can also set the image to a custom image or to one of the images stored in the Office system (described following this table).
`RibbonButtonGroup`	This is a container control that represents a group of buttons. It can contain `RibbonButton`, `RibbonGallery`, `RibbonMenu`, `RibbonSplitButton`, and `RibbonToggleButton` controls.
`RibbonCheckBox`	A check box control with a `Click` event and a `Checked` property.

continues

(continued)

CONTROL	DESCRIPTION
`RibbonComboBox`	A combo box (combined text entry with drop-down list of items). Use the `Items` property for items, the `Text` property for the entered text, and the `TextChanged` event to respond to changed.
`RibbonDropDown`	A container that can contain `RibbonDropDownItem` and `RibbonButton` items, in `Items` and `Buttons` properties respectively. The buttons and items are formatted into a drop-down list. You use the `SelectionChanged` event to respond to interaction.
`RibbonEditBox`	A text box that users can use to enter or edit text in the `Text` property. This control has a `TextChanged` event.
`RibbonGallery`	As with `RibbonDropDown`, this control can contain `RibbonDropDownItem` and `RibbonButton` items, in `Items` and `Buttons` properties respectively. This control uses `Click` and `ButtonClick` events rather than the `SelectionChanged` event that `RibbonDropDown` has.
`RibbonLabel`	Simple text display, set with the `Label` property.
`RibbonMenu`	A pop-up menu that you can populate with other controls, such as `RibbonButton` and nested `RibbonMenu` controls, when it is open in design view. Handle events for the items on the menu.
`RibbonSeparator`	A simple separator used to customize control layout in groups.
`RibbonSplitButton`	Control that combines a `RibbonButton` or `RibbonToggleButton` with a `RibbonMenu`. Set the button style with `ButtonType`, which can be `RibbonButtonType.Button` or `RibbonButtonType.ToggleButton`. Use the `Click` event for the main button or individual button `Click` events in the menu to respond to interaction.
`RibbonToggleButton`	A button that can be in a selected or unselected state, as indicated by the `Checked` property. This control also has a `Click` event.

You can also set the `DialogBoxLauncher` property of a group so that an icon appears in the bottom right of the group. You can use this to display a dialog box as its name suggests, or to open a task pane, or to perform any other action you want. You add or remove this icon through the GroupView Tasks menu, as shown in Figure 49-9, which also shows some of the other controls in the previous table as they appear on a ribbon in design view.

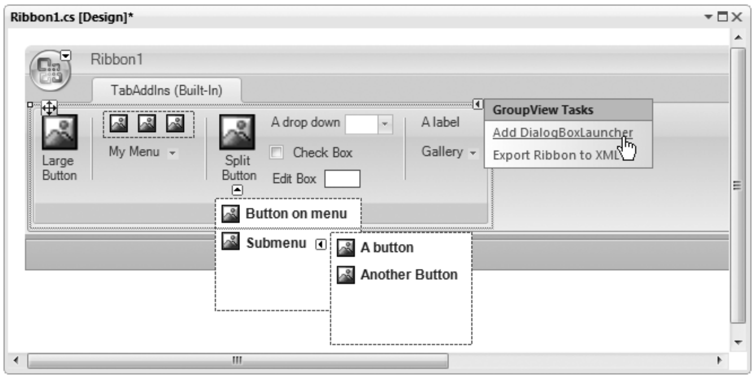
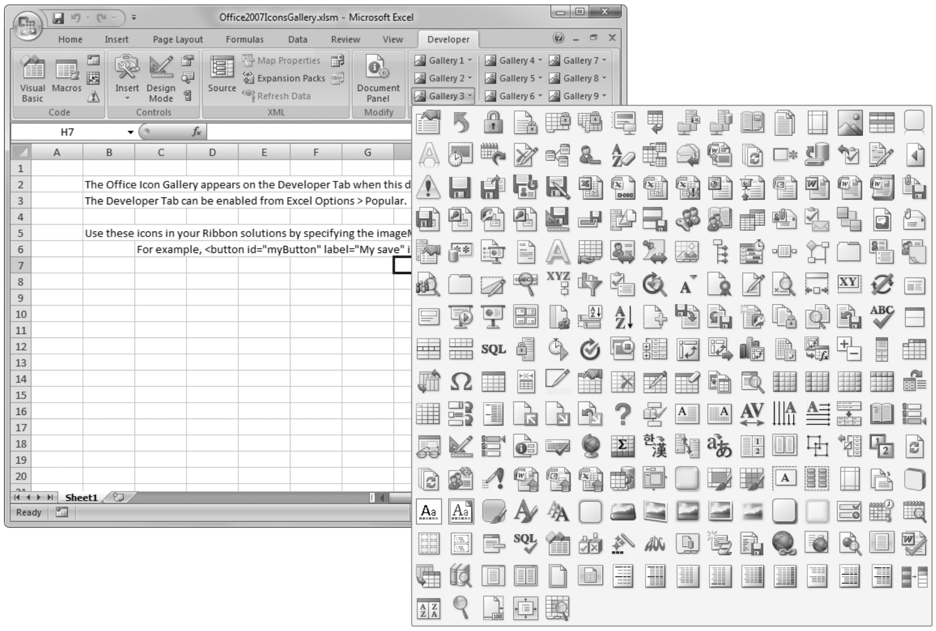


FIGURE 49-9

To set the image for a control, for example a `RibbonButton` control, you can either set the `Image` property to a custom image and `ImageName` to a name for the image (so that you can optimize image loading in an `OfficeRibbon.LoadImage` event hander), or you can use one of the built-in Office images. To do this, you set the `OfficeImageId` property to the ID of the image.

There are many images that you can use; you can download a spreadsheet that lists them from `www.microsoft.com/downloads/details.aspx?familyid=12b99325-93e8-4ed4-8385-74d0f7661318&displaylang=en`. Figure 49-10 shows a sample.

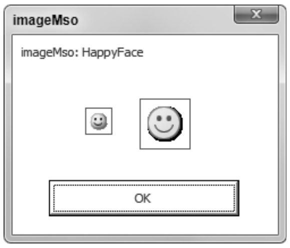


The Office Icon Gallery appears on the Developer Tab when this d
The Developer Tab can be enabled from Excel Options > Popular.
Use these icons in your Ribbon solutions by specifying the imageM
For example, <button id="myButton" label="My save" i

FIGURE 49-10

Figure 49-10 shows the Developer ribbon tab, which you can enable through the Office button, in the Excel Options dialog box, on the Popular tab.

When you click on an image, a dialog box appears to tell you what the image ID is, as shown in Figure 49-11.



imageMso: HappyFace

FIGURE 49-11

The ribbon designer is extremely flexible, and you can provide pretty much any functionality that you would expect to find on an Office ribbon. However, if you want to customize your UI further, then you will want to use action and task panes, as you can create any UI and functionality you like there.

Action Panes and Custom Task Panes

You can use action and task panes to display content that is docked in the task pane area of the Office application interface. Task panes are used in application-level add-ins, and action panes are used in document-level customizations. Both task and action panes must inherit from `UserControl` objects, which means that you create a UI by using Windows Forms. You can also use a WPF UI if you host a WPF form in an `ElementHost` control on the `UserControl`.

To add an action pane to a document in a document-level customization, you add an instance of the action pane class to the `Controls` collection of the `ActionsPane` property of the document. For example:

```
public partial class ThisWorkbook
{
   private UserControl1 actionsPane;
```

```
      private void ThisWorkbook_Startup(object sender, System.EventArgs e)
      {
         Workbook wb = Globals.Factory.GetVstoObject(this.Application.ActiveWorkbook);
         wb.AcceptAllChanges();
         actionsPane = new UserControl1();
         this.ActionsPane.Controls.Add(actionsPane);
      }
      ...
   }
```

This code adds the actions pane when the document (in this case an Excel workbook) is loaded. You can also do this in, for example, a ribbon button event handler.

Custom task panes are added through the `ThisAddIn.CustomTaskPanes.Add()` method property in application-level add-in projects. This method also allows you to name the task window. For example:

```
   public partial class ThisAddIn
   {
      Microsoft.Office.Tools.CustomTaskPane taskPane;
      private void ThisAddIn_Startup(object sender, System.EventArgs e)
      {
         taskPane = this.CustomTaskPanes.Add(new UserControl1(), "My Task Pane");
         taskPane.Visible = true;
      }
      ...
   }
```

Note that the `Add()` method returns an object of type `Microsoft.Office.Tools.CustomTaskPane`. You can access the user control itself through the `Control` property of this object. You can also use other properties exposed by this type — for example, the `Visible` property as shown in the previous code — to control the task pane.

At this point, it is worth mentioning a slightly unusual feature of Office applications, and in particular, a difference between Word and Excel. For historical reasons, although both Word and Excel are MDI applications, the way in which these applications host documents is different. In Word, every document has a unique parent window. In Excel, every document shares the same parent window.

When you call the `CustomTaskPanes.Add()` method, the default behavior is to add the task pane to the currently active window. In Excel, this means that every document will display the task pane, as the same parent window is used for all of them. In Word, the situation is different. If you want the task pane to appear for every document, then you must add it to every window that contains a document.

To add the task pane to a specific document, you pass an instance of the `Microsoft.Office.Interop.Word.Window` class to the `Add()` method as a third parameter. You can obtain the window associated with a document through the `Microsoft.Office.Interop.Word.Document.ActiveWindow` property.

In the next section, you will see how to do this in practice.

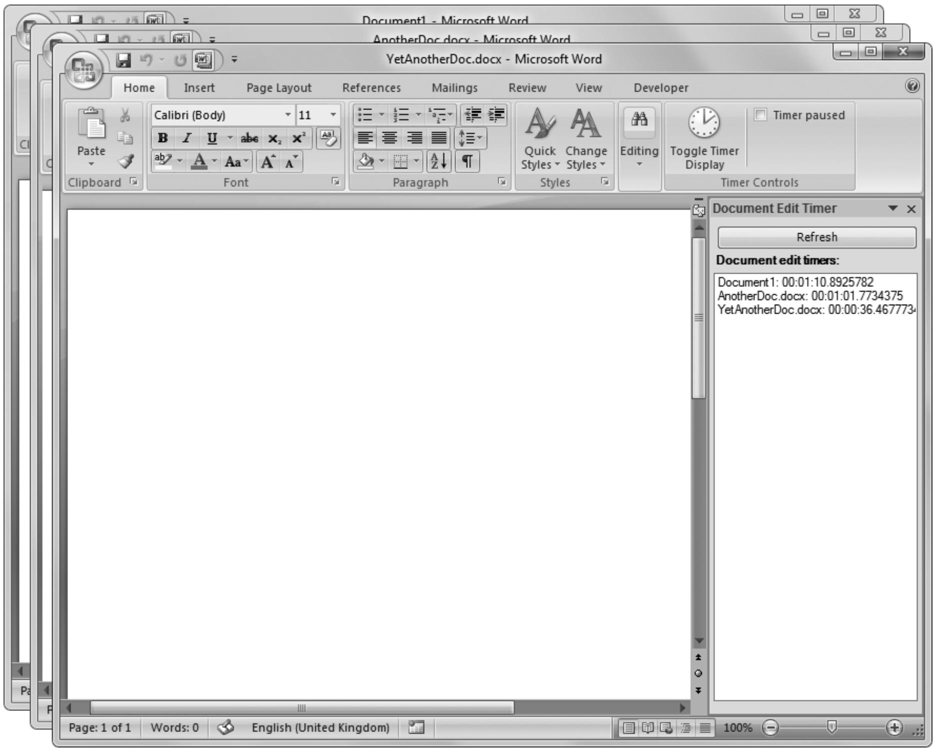
EXAMPLE APPLICATION

As mentioned in previous sections, the example code for this chapter includes an application called `WordDocEditTimer`, which maintains a list of edit times for Word documents. In this section, we examine the code for this application in detail, as it illustrates everything you've read about so far and includes some useful tips.

The general operation of this application is that whenever a document is created or loaded, a timer is started, which is linked to the document name. If you close a document, then the timer for that document pauses. If you open a document that has previously been timed, then the timer resumes. Also, if you use Save As to save a document with a different filename, then the timer is updated to use the new filename.

This application is a Word application-level add-in, and uses a custom task pane and a ribbon menu. The ribbon menu contains a button that you can use to turn the task pane on and off and a check box that enables you to pause the timer for the currently active document. The group containing these controls is appended to the Home ribbon tab. The task pane displays a list of active timers.

This user interface is shown in Figure 49-12.



Document edit timers:

```
Document1: 00:01:10.8925782
AnotherDoc.docx: 00:01:01.7734375
YetAnotherDoc.docx: 00:00:36.467773
```

FIGURE 49-12

Timers are maintained through the `DocumentTimer` class:

Available for download on Wrox.com

```
public class DocumentTimer
{
   public Word.Document Document { get; set; }
   public DateTime LastActive { get; set; }
   public bool IsActive { get; set; }
   public TimeSpan EditTime { get; set; }
}
```

code snippet DocumentTimer.cs

This keeps a reference to a `Microsoft.Office.Interop.Word.Document` interface as well as the total edit time, whether the timer is active, and the time it last became active. The `ThisAddIn` class maintains a collection of these objects, associated with document names:

Available for download on Wrox.com

```
public partial class ThisAddIn
{
   private Dictionary<string, DocumentTimer> documentEditTimes;
```

code snippet ThisAddIn.cs

Each timer can therefore be located by document reference or document name. This is necessary because document references allow you to keep track of document name changes (there is no event that you can use to monitor this), and document names allow you to keep track of closed and reopened documents.

The ThisAddIn class also maintains a list of CustomTaskPane objects (as noted earlier, one is required for each window in Word):

```
private List<Tools.CustomTaskPane> timerDisplayPanes;
```

When the add-in starts, the ThisAddIn_Startup() method performs several tasks. First, it initializes the two collections:

```
private void ThisAddIn_Startup(object sender, System.EventArgs e)
{
   // Initialize timers and display panels
   documentEditTimes = new Dictionary<string, DocumentTimer>();
   timerDisplayPanes = new List<Microsoft.Office.Tools.CustomTaskPane>();
```

Next, it adds several event handlers through the ApplicationEvents4_Event interface:

```
   // Add event handlers
   Word.ApplicationEvents4_Event eventInterface = this.Application;
   eventInterface.DocumentOpen += new Microsoft.Office.Interop.Word
      .ApplicationEvents4_DocumentOpenEventHandler(
         eventInterface_DocumentOpen);
   eventInterface.NewDocument += new Microsoft.Office.Interop.Word
      .ApplicationEvents4_NewDocumentEventHandler(
         eventInterface_NewDocument);
   eventInterface.DocumentBeforeClose += new Microsoft.Office.Interop.Word
      .ApplicationEvents4_DocumentBeforeCloseEventHandler(
         eventInterface_DocumentBeforeClose);
   eventInterface.WindowActivate += new Microsoft.Office.Interop.Word
      .ApplicationEvents4_WindowActivateEventHandler(
         eventInterface_WindowActivate);
```

These event handlers are used to monitor documents as they are opened, created, and closed, and also to ensure that the Pause check box is kept up-to-date on the ribbon. This latter functionality is achieved by keeping track of window activations with the WindowActivate event.

The last task performed in this event handler is to start monitoring the current document and add the custom task panel to the window containing the document:

```
   // Start monitoring active document
   MonitorDocument(this.Application.ActiveDocument);
   AddTaskPaneToWindow(this.Application.ActiveDocument.ActiveWindow);
}
```

The MonitorDocument() utility method adds a timer for a document:

```
internal void MonitorDocument(Word.Document Doc)
{
   // Monitor doc
   documentEditTimes.Add(Doc.Name, new DocumentTimer
   {
      Document = Doc,
      EditTime = new TimeSpan(0),
      IsActive = true,
      LastActive = DateTime.Now
   });
}
```

This method simply creates a new DocumentTimer for the document. The DocumentTimer references the document, has zero edit time, is active, and was made active at the current time. It then adds this timer to the documentEditTimes collection and associates it with the document name.

The AddTaskPaneToWindow() method adds the custom task pane to a window. This method starts by checking the existing task panes to ensure that there isn't one in the window already. Also, one other strange feature of Word is that if you immediately open an old document after loading the application, the default Document1 document vanishes, without raising a close event. This can lead to an exception being raised when the window for the task pane that was in the document is accessed, so the method also checks for the `ArgumentNullException` that indicates this:

```
private void AddTaskPaneToWindow(Word.Window Wn)
{
   // Check for task pane in window
   Tools.CustomTaskPane docPane = null;
   Tools.CustomTaskPane paneToRemove = null;
   foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
   {
      try
      {
         if (pane.Window == Wn)
         {
            docPane = pane;
            break;
         }
      }
      catch (ArgumentNullException)
      {
         // pane.Window is null, so document1 has been unloaded.
         paneToRemove = pane;
      }
   }
```

If an exception is thrown, then the offending task pane is removed from the collection:

```
   // Remove pane if necessary
   timerDisplayPanes.Remove(paneToRemove);
```

If no task pane was found for the window, then the method finishes by adding one:

```
   // Add task pane to doc
   if (docPane == null)
   {
      Tools.CustomTaskPane pane = this.CustomTaskPanes.Add(
         new TimerDisplayPane(documentEditTimes),
         "Document Edit Timer",
         Wn);
      timerDisplayPanes.Add(pane);
      pane.VisibleChanged +=
         new EventHandler(timerDisplayPane_VisibleChanged);
   }
}
```

The added task pane is an instance of the `TimerDisplayPane` class. You will look at this class shortly. It is added with the name "Document Edit Timer." Also, an event handler is added for the `VisibleChanged` event of the `CustomTaskPane` that you obtain after calling the `CustomTaskPanes.Add()` method. This enables you to refresh the display when it first appears:

```
private void timerDisplayPane_VisibleChanged(object sender, EventArgs e)
{
   // Get task pane and toggle visibility
   Tools.CustomTaskPane taskPane = (Tools.CustomTaskPane)sender;
   if (taskPane.Visible)
   {
      TimerDisplayPane timerControl = (TimerDisplayPane)taskPane.Control;
      timerControl.RefreshDisplay();
   }
}
```

The `TimerDisplayPane` class exposes a `RefreshDisplay()` method that is called in the preceding code. This method, as its name suggests, refreshes the display of the `timerControl` object.

Next, there is the code that ensures that all documents are monitored. First, when a new document is created, the `eventInterface_NewDocument()` event handler is called, and the document is monitored by calling the `MonitorDocument()` and `AddTaskPaneToWindow()` methods, which you've already seen.

```
private void eventInterface_NewDocument(Word.Document Doc)
{
   // Monitor new doc
   MonitorDocument(Doc);
   AddTaskPaneToWindow(Doc.ActiveWindow);
```

This method also clears the Pause check box in the ribbon menu as new documents start with the time running. This is achieved through a utility method, `SetPauseStatus()`, which is defined on the ribbon:

```
   // Set checkbox
   Globals.Ribbons.TimerRibbon.SetPauseStatus(false);
}
```

Just before a document is closed, the `eventInterface_DocumentBeforeClose()` event handler is called. This method freezes the timer for the document, updates the total edit time, clears the `Document` reference, and removes the task pane from the document window (with `RemoveTaskPaneFromWindow()`, detailed shortly) before the document is closed.

```
private void eventInterface_DocumentBeforeClose(Word.Document Doc,
   ref bool Cancel)
{
   // Freeze timer
   documentEditTimes[Doc.Name].EditTime += DateTime.Now
      - documentEditTimes[Doc.Name].LastActive;
   documentEditTimes[Doc.Name].IsActive = false;
   documentEditTimes[Doc.Name].Document = null;
   // Remove task pane
   RemoveTaskPaneFromWindow(Doc.ActiveWindow);
}
```

When a document is opened, the `eventInterface_DocumentOpen()` method is called. There is a little more work to be done here, as before monitoring the document, the method must determine whether a timer already exists for the document by looking at its name:

```
private void eventInterface_DocumentOpen(Word.Document Doc)
{
   if (documentEditTimes.ContainsKey(Doc.Name))
   {
      // Monitor old doc
      documentEditTimes[Doc.Name].LastActive = DateTime.Now;
      documentEditTimes[Doc.Name].IsActive = true;
      documentEditTimes[Doc.Name].Document = Doc;
      AddTaskPaneToWindow(Doc.ActiveWindow);
   }
```

If the document isn't already being monitored, then a new monitor is configured as for a new document:

```
   else
   {
      // Monitor new doc
      MonitorDocument(Doc);
      AddTaskPaneToWindow(Doc.ActiveWindow);
   }
}
```

The `RemoveTaskPaneFromWindow()` method is used to remove the task pane from a window. The code for this method first checks that a task pane exists for the specified window:

```
private void RemoveTaskPaneFromWindow(Word.Window Wn)
{
   // Check for task pane in window
   Tools.CustomTaskPane docPane = null;
```

```
         foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
         {
            if (pane.Window == Wn)
            {
               docPane = pane;
               break;
            }
         }
```

If a task window is found, then it is removed by calling the `CustomTaskPanes.Remove()` method. It is also removed from the local collection of task pane references.

```
         // Remove document task pane
         if (docPane != null)
         {
            this.CustomTaskPanes.Remove(docPane);
            timerDisplayPanes.Remove(docPane);
         }
      }
```

The last event handler in this class is `eventInterface_WindowActivate()`, called when a window is activated. This method gets the timer for the active document (testing first to see if there is one, as new documents may not have had one added at the time this method is called) and sets the check box on the ribbon menu so that the check box is kept updated for the document:

```
      private void eventInterface_WindowActivate(Word.Document Doc,
         Word.Window Wn)
      {
         if (documentEditTimes.ContainsKey(this.Application.ActiveDocument.Name))
         {
            // Ensure pause checkbox in ribbon is accurate, start by getting timer
            DocumentTimer documentTimer =
               documentEditTimes[this.Application.ActiveDocument.Name];
            // Set checkbox
            Globals.Ribbons.TimerRibbon.SetPauseStatus(!documentTimer.IsActive);
         }
      }
```

The code for `ThisAddIn` also includes two utility methods. The first of these, `ToggleTaskPaneDisplay()`, is used to show or hide the display of the task pane for the currently active document by setting the `CustomTaskPane.Visible` property.

```
      internal void ToggleTaskPaneDisplay()
      {
         // Ensure window has task window
         AddTaskPaneToWindow(this.Application.ActiveDocument.ActiveWindow);
         // toggle document task pane
         Tools.CustomTaskPane docPane = null;
         foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
         {
            if (pane.Window == this.Application.ActiveDocument.ActiveWindow)
            {
               docPane = pane;
               break;
            }
         }
         docPane.Visible = !docPane.Visible;
      }
```

The `ToggleTaskPaneDisplay()` method shown in the preceding code is called by event handlers on the ribbon control, as you will see shortly.

Finally, the class has another method that is called from the ribbon menu, which enables ribbon controls to pause or resume the timer for a document:

```
    internal void PauseOrResumeTimer(bool pause)
    {
       // Get timer
       DocumentTimer documentTimer =
          documentEditTimes[this.Application.ActiveDocument.Name];
       if (pause && documentTimer.IsActive)
       {
          // Freeze timer
          documentTimer.EditTime += DateTime.Now - documentTimer.LastActive;
          documentTimer.IsActive = false;
       }
       else if (!pause && !documentTimer.IsActive)
       {
          // Resume timer
          documentTimer.IsActive = true;
          documentTimer.LastActive = DateTime.Now;
       }
    }
 }
```

The only other code in this class definition is an empty event handler for `Shutdown`, and the VSTO-generated code to hook up the `Startup` and `Shutdown` event handlers.

Next, the ribbon in the project, `TimerRibbon`, is laid out, as shown in Figure 49-13.

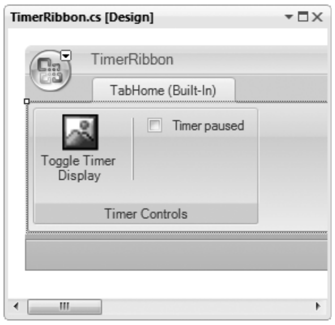


FIGURE 49-13

This ribbon contains a `RibbonButton`, a `RibbonSeparator`, a `RibbonCheckBox`, and a `DialogBoxLauncher`. The button uses the large display style, and has an `OfficeImageId` of `StartAfter Previous`, which displays the clock face shown in Figure 49-13. (These images are not visible at design time.) The ribbon uses the TabHome tab type, which causes its contents to be appended to the Home tab.

The ribbon has three event handlers, each of which calls on one of the utility methods in `ThisAddIn` described earlier:

Available for download on Wrox.com

```
private void group1_DialogLauncherClick(object sender,
   RibbonControlEventArgs e)
{
   // Show or hide task pane
   Globals.ThisAddIn.ToggleTaskPaneDisplay();
}
private void pauseCheckBox_Click(object sender, RibbonControlEventArgs e)
{
   // Pause timer
   Globals.ThisAddIn.PauseOrResumeTimer(pauseCheckBox.Checked);
}
private void toggleDisplayButton_Click(object sender,
   RibbonControlEventArgs e)
{
   // Show or hide task pane
   Globals.ThisAddIn.ToggleTaskPaneDisplay();
}
```

code snippet TimerRibbon.cs

The ribbon also includes its own utility method, `SetPauseStatus()`, which as you saw earlier is called by code in `ThisAddIn` to select or clear the check box:

```
internal void SetPauseStatus(bool isPaused)
{
   // Ensure checkbox is accurate
   pauseCheckBox.Checked = isPaused;
}
```

The other component in this solution is the `TimerDisplayPane` user control that is used in the task pane. The layout of this control is shown in Figure 49-14.

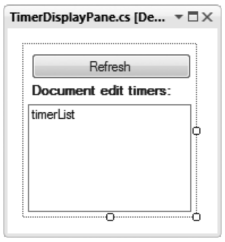


FIGURE 49-14

This control includes a button, a label, and a list box — not the most exciting of displays, although it would be simple enough to replace it with, for example, a prettier WPF control.

The code for the control keeps a local reference to the document timers, which is set in the constructor:

Available for download on Wrox.com

```
public partial class TimerDisplayPane : UserControl
{
   private Dictionary<string, DocumentTimer> documentEditTimes;
   public TimerDisplayPane()
   {
      InitializeComponent();
   }
   public TimerDisplayPane(Dictionary<string, DocumentTimer>
      documentEditTimes): this()
   {
      // Store reference to edit times
      this.documentEditTimes = documentEditTimes;
   }
```

code snippet TimerDisplayPane.cs

The button event handler calls the `RefreshDisplay()` method to refresh the timer display:

```
   private void refreshButton_Click(object sender, EventArgs e)
   {
      RefreshDisplay();
   }
```

The `RefreshDisplay()` method is also called from `ThisAddIn`, as you saw earlier. It is a surprisingly complicated method considering what it does. It also checks the list of monitored documents against the list of loaded documents and corrects any problems. This sort of code is often necessary in VSTO applications, as the interface with the COM Office object model occasionally doesn't work quite as it should. The rule of thumb here is to code defensively.

The method starts by clearing the current list of timers in the `timerList` list box:

```
   internal void RefreshDisplay()
   {
      // Clear existing list
      this.timerList.Items.Clear();
```

Next, the monitors are checked. The method iterates through each document in the `Globals.ThisAddIn.Application.Documents` collection and determines if the document is monitored, unmonitored, or monitored but has had a name change since the last refresh.

Finding monitored documents simply involves checking the document name against the document names in the `documentEditTimes` collection of keys:

```
      // Ensure all docs are monitored
      foreach (Word.Document doc in Globals.ThisAddIn.Application.Documents)
      {
         bool isMonitored = false;
         bool requiresNameChange = false;
         DocumentTimer oldNameTimer = null;
         string oldName = null;
```

```
   foreach (string documentName in documentEditTimes.Keys)
   {
      if (doc.Name == documentName)
      {
         isMonitored = true;
         break;
      }
```

If the names don't match, then the document references are compared, which enables you to detect name changes to documents, as shown in the following code:

```
      else
      {
         if (documentEditTimes[documentName].Document == doc)
         {
            // Monitored, but name changed!
            oldName = documentName;
            oldNameTimer = documentEditTimes[documentName];
            isMonitored = true;
            requiresNameChange = true;
            break;
         }
      }
   }
```

For unmonitored documents, a new monitor is created:

```
   // Add monitor if not monitored
   if (!isMonitored)
   {
      Globals.ThisAddIn.MonitorDocument(doc);
   }
```

Whereas documents with name changes are re-associated with the monitor used for the old named document:

```
   // Rename if necessary
   if (requiresNameChange)
   {
      documentEditTimes.Remove(oldName);
      documentEditTimes.Add(doc.Name, oldNameTimer);
   }
}
```

After reconciling the document edit timers, a list is generated. This code also detects whether referenced documents are still loaded, and pauses the timer for documents that aren't by setting the `IsActive` property to `false`. Again, this is defensive programming.

```
// Create new list
foreach (string documentName in documentEditTimes.Keys)
{
   // Check to see if doc is still loaded
   bool isLoaded = false;
   foreach (Word.Document doc in
      Globals.ThisAddIn.Application.Documents)
   {
      if (doc.Name == documentName)
      {
         isLoaded = true;
         break;
      }
```

```
            }
            if (!isLoaded)
            {
               documentEditTimes[documentName].IsActive = false;
               documentEditTimes[documentName].Document = null;
            }
```

For each monitor, a list item is added to the list box that includes the document name and its total edit time:

```
            // Add item
            this.timerList.Items.Add(string.Format("{0}: {1}", documentName,
               documentEditTimes[documentName].EditTime +
               (documentEditTimes[documentName].IsActive ?
               (DateTime.Now — documentEditTimes[documentName].LastActive):
               new TimeSpan(0))));
         }
      }
   }
```

This completes the code in this example. This example has shown you how to use ribbon and task pane controls and how to maintain task panes in multiple Word documents. It has also illustrated many of the techniques covered earlier in the chapter.

SUMMARY

In this chapter you have learned how to use VSTO to create managed solutions for Office products.

In the first part of this chapter, you learned about the general structure of VSTO projects and the project types that you can create. You also saw the features that you can use to make VSTO programming easier.

In the next section, you looked, in great depth, at some of the features available in VSTO solutions and you saw how communication with the Office object model is achieved. You also looked at the namespaces and types available in VSTO and learned how to use those types to implement a variety of functionality. Then, you explored some of the code features of VSTO projects and how to use these features to get the effect you want.

After this, you moved on to the more practical side of things. You learned how add-ins are managed in Office applications and how to interact with the Office object model. You also saw how to customize the UI of your applications with ribbon menus, task panes, and action panes.

Finally, you explored an example application that illustrated the UI and interaction techniques that you learned earlier. The example contained a lot of code, but it included useful techniques, including how to manage task panes in multiple Word document windows.