Microsoft SharePoint

Building Office 2003 Solutions, Second Edition

Scot P. Hillier

Apress®

Microsoft SharePoint: Building Office 2003 Solutions, Second Edition

ISBN (pbk): 1-59059-575-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jim Sumser
Technical Reviewer: Judith M. Myerson
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editors: Rebecca Rider, Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Lori Bring
Indexer: Tim Tate
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 10

Visual Studio 2005 and the Microsoft Office System

Throughout this book, I have used Visual Studio 2003 (VS2003) to create Web Parts and Office solutions. However, VS2003 was not really designed with SharePoint and Office System projects in mind. You can certainly see evidence of this in the fact that there is no inherent support for creating Web Parts in VS2003 as well as in the amount of coding necessary to create a Smart Document for Office.

As this edition goes to press, Microsoft is preparing to release Visual Studio 2005 (VS2005), which contains tools, enhancements, and project types of interest to the SharePoint developer. Additionally, VS2005 is designed to be used with the .NET Framework 2.0, which delivers significant new support for Web Parts that can be used outside of the SharePoint environment. Although this chapter is written against the Beta 2 release of VS2005, I felt the integration with the Office System justified an early look. I just have to make the standard disclaimer that some of this information may change by the time the final product is released.

As of this writing, you can get a copy of VS2005 Beta 2 from Microsoft by visiting the Visual Studio 2005 home page at `http://lab.msdn.microsoft.com/vs2005`. On the VS2005 home page, you can download one of the many editions of Visual Studio. The Express editions are intended to be lightweight versions of Visual Studio targeted at novice developers. These editions include versions for web development, VB .NET, C#, C++, and J#. Additionally, you can download an Express version of SQL Server 2005 to use in conjunction with the development environment. Professional developers will not likely use any of the Express versions; instead, they will make use of Visual Studio Team System (VSTS).

VSTS is intended to be a single consolidated environment that supports all members of the software development team. VSTS has separate editions for architects, developers, testers, and project managers. Each of these editions is intended to provide the toolset necessary for a particular role. Architects, for example, would have access to design and modeling tools. Developers would utilize the integrated debugging environment along with source code control. Testers would make use of unit testing and performance tools, while project managers would use Microsoft Project and Windows SharePoint Services to manage the software life cycle.

Complete coverage of VSTS is well beyond the scope of this book, but I do want to talk about things that are of particular importance to the SharePoint developer. Therefore, I have set up a development environment that includes VSTS and Microsoft Office 2003 on a Windows XP client. Using this simple setup, we can investigate two key technologies: the ASP.NET 2.0 Web Parts Framework and the Visual Studio 2005 Tools for Office (VSTO).

The ASP.NET 2.0 Web Parts Framework

The power of SharePoint as a solution platform comes in no small measure from its support for Web Parts. The Web Parts framework built into Windows SharePoint Services (WSS) provides a consistent environment for both developer and user. Standard interfaces, attributes, and deployment models make Web Part construction straightforward, while standard interface elements to add, remove, and modify Web Parts make them easy to customize. The only drawback to using Web Parts is that a complete installation of WSS is required to utilize the framework.

Beginning with the next release of the .NET Framework and Visual Studio 2005, developers will no longer be limited to using Web Parts solely within SharePoint environments. This is because Microsoft has built the Web Parts framework into the .NET Framework class library. The set of classes that implement the framework are known collectively as the ASP.NET 2.0 Web Parts Framework, and they allow you to can create and deploy Web Parts for custom applications as well as the next version of SharePoint technologies. Although the next version of SharePoint will not be available until late 2006, you can begin to get familiar with the framework upon which it will be based now.

Understanding the Web Parts Control Set

The .NET Framework classes that implement the Web Parts framework are intended to be used within ASP.NET applications. Before you can utilize any Web Parts, however, you must use several of the .NET classes to implement the basic functions of the framework. These basic functions provide support for zones, layouts, and property management. In VS2005, all of the required classes are implemented as server controls known collectively as the Web Parts control set. When you create a new ASP.NET application, these controls appear automatically in the Visual Studio toolbox as shown in Figure 10-1.

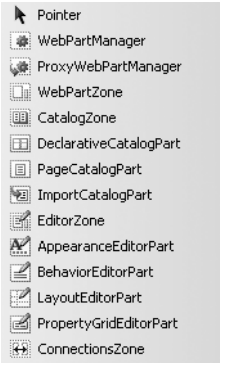

Figure 10-1. *The Web Parts control set*

Every ASP.NET page that contains Web Parts must include a single `WebPartManager` control. This control must be dragged from the toolbox and placed at the top of the page. The `WebPartManager` control provides much of the foundational functionality of the Web Parts framework, but it is not visible at runtime. Once it is in place, however, you can add other controls that implement visible elements.

After adding a `WebPartManager` control, you can use the `WebPartZone` control to define zones within the page. These zones work exactly like the zones in SharePoint; they define areas where you can add Web Parts to the page. In fact, Visual Studio will allow you to use any standard control as a Web Part once the `WebPartManager` and `WebPartZone` controls are in place.

Follow these steps to use a standard control as a Web Part:

1. Start Visual Studio 2005 and select File ➤ New ➤ Web Site from the main menu.
2. In the New Web Site dialog, select the ASP.NET Web Site template.
3. In the Location drop-down list, select File System.
4. Click the Browse button.
5. In the Choose Location dialog, select a location in the file system tree to create the new web site.
6. Create a new folder and name it **SimpleSite**.
7. Click the Open button to return to the New Web Site dialog.
8. In the New Web Site dialog, click the OK button to create the new web site.
9. In the Solution Explorer, select the `Default.aspx` file and click the View Designer button.
10. Drag a `WebPartManager` control from the toolbox and place it at the top of the `Default.aspx` page.
11. Drag a `WebPartZone` control from the toolbox and place it directly below the `WebPartManager` control.
12. Expand the Standard control set in the toolbox and drag a `Label` control into the `WebPartZone` control.
13. Select Debug ➤ Start Without Debugging from the main menu.

When you run this simple example, you will see the `Label` control visible within the Web Part zone. You will also notice that a drop-down menu is available that allows you to minimize or close the Web Part. However, there is no capability as of yet to change the layout or appearance of the page. In order to implement that capability, you must write some code to change the display mode of the page and add some additional controls to the page.

Changing the display mode of a page permits dragging Web Parts between zones, changing Web Part properties, connecting Web Parts, and adding new parts to the page. Changing the display mode is a simple matter of setting the `DisplayMode` property of the `WebPartManager` in code. However, each mode also requires one or more additional controls to implement the user interface necessary to modify the page layout or Web Part properties.

The `EditorZone` control creates a special zone on the web page where you can place additional controls that allow the page or Web Parts to be modified. Once an `EditorZone` is placed, you may add additional `AppearanceEditorPart`, `LayoutEditorPart`, `BehaviorEditorPart`, or `PropertyGridEditorPart` controls to the zone. The `EditorZone` and its associated controls remain invisible until the `DisplayMode` is changed to reveal them.

The `CatalogZone` control creates a special zone on the web page where you can place additional controls that allow new Web Parts to be added. Once a `CatalogZone` is placed, you may add additional `DeclarativeCatalogPart`, `PageCatalogPart`, or `ImportCatalogPart` controls to the zone. The `CatalogZone` and its associated controls remain invisible until the `DisplayMode` is changed to reveal them.

The `ConnectionsZone` control creates a special zone on the web page where you can make connections between Web Parts. Just like in SharePoint, you can pass information between Web Parts to create more complicated user interfaces. Table 10-1 lists the settings for the `DisplayMode` property, its resulting effect on the web page, and the associated controls that allow editing or managing Web Parts.

Table 10-1. *DisplayMode Settings*

Value	Description	Associated Controls
`WebPartManager.BrowseDisplayMode`	Displays the page normally.	
`WebPartManager.DesignDisplayMode`	Displays the Web Part zones. Allows Web Parts to be dragged between zones.	`WebPartZone`
`WebPartManager.EditDisplayMode`	Displays the Web Part zones and editing controls. Allows Web Parts to be dragged between zones and Web Part properties to be edited.	`EditorZone`, `AppearanceEditorPart`, `LayoutEditorPart`, `BehaviorEditorPart`, `PropertyGridEditorPart`
`WebPartManager.CatalogDisplayMode`	Displays the Web Parts zones and catalog controls. Allows Web Parts to be dragged between zones and new Web Parts to be added to the page.	`CatalogZone`, `DeclarativeCatalogPart`, `PageCatalogPart`, `ImportCatalogPart`
`WebPartManager.ConnectDisplayMode`	Displays the Web Part zones. Allows Web Parts to be connected.	`ConnectionsZone`

Building Custom Web Parts

Although you can use any standard control as a Web Part if it is supported by the Web Parts control set, most of the time you will find that you still need to build your own custom Web Parts from scratch. First of all, standard controls are limited in functionality and not easily extended. Second, the standard controls will not work as Web Parts in the next version of SharePoint. The good news, however, is that building a Web Part in ASP.NET 2.0 is very similar to building one in SharePoint 2003.

Creating a custom Web Part in ASP.NET 2.0 begins by inheriting from the `WebPart` class, in much the same way as in SharePoint 2003. The big difference is that the base class for Web Parts in SharePoint 2003 derives from `Microsoft.SharePoint.WebPartPages.WebPart`, whereas the base class for ASP.NET 2.0 Web Parts and the next release of SharePoint is `System.Web.UI.➥ WebControls.WebParts.WebPart`. Although Microsoft does promise backward compatibility with Web Parts built on the `Microsoft.SharePoint` namespace, all future development will use the new namespace of ASP.NET 2.0.

You begin the definition of a new Web Part by creating a new Class Library project in C# or VB .NET. Once the project is created, you must set a reference to the `System.Web` namespace, which contains the `WebPart` base class. Once the reference is set, you may then set up the class to inherit from the base class. As an example, I'll build two image viewer parts that will contain

a property for specifying a URL for an image file. Listing 10-1 shows the foundational code for the Web Part built in C#, and Listing 10-2 shows the code in VB .NET.

Listing 10-1. *Starting a Web Part in C#*

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls.WebParts;

namespace CViewer
{
    public class Frame:WebPart
    {

    }
}
```

Listing 10-2. *Starting a Web Part in VB .NET*

```
mports System
Imports System.Web
Imports System.Web.UI.WebControls.WebParts

Public Class Frame
    Inherits WebPart

End Class
```

ASP.NET 2.0 Web Parts are still based on the concept of a server control, just like SharePoint 2003 Web Parts. Therefore, they have essentially the same life cycle as I outlined in Chapter 5. There are differences, however, in the names of the methods and attributes used within the class module. For example, ASP.NET 2.0 Web Parts have a `RenderContents` method instead of a `RenderWebPart` method. Aside from the name, everything else about these methods is the same. You still use an `HtmlTextWriter` to generate the output that will be displayed to the user. Although the names of some of the methods are different, some are still the same. For example, you can still override the `CreateChildControls` method to add your own controls to the Web Part.

Creating properties for Web Parts in ASP.NET 2.0 is also nearly identical to SharePoint 2003. Again, the only real difference is in the naming; ASP.NET 2.0 attributes have different names than their SharePoint 2003 counterparts. For example, declaring that a property is `WebBrowsable` will allow its properties to be edited in the `PropertyGridEditorPart`, which I'll cover later in the chapter. Listing 10-3 shows the viewer Web Part in C#, and Listing 10-4 shows the code in VB .NET.

Listing 10-3. *The Completed Web Part in C#*

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls.WebParts;
```

```
namespace CViewer
{

    public class Frame:WebPart
    {
        private string _url =
            "http://www.datalan.com/images/DLlogo2.jpg";
        public string Url
        {
            get{return _url;}
            set{_url = value;}
        }

        protected override void RenderContents(HtmlTextWriter writer)
        {
            writer.Write("<IMG SRC=\"" +
                Url + "\" HEIGHT=\"60px\" WIDTH=\"202px\">");
        }

    }
}
```

Listing 10-4. *The Completed Web Part in VB .NET*

```
Imports System
Imports System.Web
Imports System.Web.UI.WebControls.WebParts

Public Class Frame
    Inherits WebPart

    Private m_URL As String = _
    "http://www.datalan.com/images/partner_microsoft_poy.gif"

    Public Property URL() As String
        Get
            Return m_URL
        End Get
        Set(ByVal value As String)
            m_URL = value
        End Set
    End Property

    Protected Overrides Sub RenderContents( _
      ByVal writer As System.Web.UI.HtmlTextWriter)
        writer.Write("<IMG SRC=""" & _
        URL & """ HEIGHT=""83px"" WIDTH=""190px"">")
    End Sub

End Class
```

Using Web Parts in a Page

One of the strengths of the SharePoint Web Part infrastructure is that it provides administration and management of Web Parts with no additional work on your part. Inside of a SharePoint site, you can view catalogs of Web Parts, import Web Parts, and modify Web Parts. In a custom application based on ASP.NET 2.0, the administrative functionality must be implemented using the Web Parts control set and writing some code into the custom web page.

While standard controls can easily be dragged from the toolbox into an existing zone, custom Web Parts cannot. Therefore, you must set a reference to the assembly containing the Web Part and register it with each web page where it will be used. This is done by including a `Register` directive in the ASP.NET code of the page. Typically, you will reference the assembly containing the custom Web Part and provide an alias for the associated namespace using the `TagPrefix` attribute. The following code shows how to register both the C# and VB .NET versions of the Web Part created earlier:

```
<%@ Register TagPrefix="csharppart" Namespace="CViewer" Assembly="CViewer" %>
<%@ Register TagPrefix="vbpart" Namespace="VBViewer" Assembly="VBViewer" %>
```

Once the assembly is registered, you may use the various catalog-management controls in the toolbox to make the custom Web Parts available in the page. You begin by dragging a `Catalog➥Zone` control from the toolbox onto the web page design surface. The `CatalogZone` acts as a host for any combination of the `DeclarativeCatalogPart`, `PageCatalogPart`, and `ImportCatalogPart`.

The `DeclarativeCatalogPart` is used to create a catalog on the page by declaring available Web Parts in ASP.NET code. The `PageCatalogPart` allows Web Parts that are closed by the user to be added back to a page, while the `ImportCatalogPart` is used to add Web Parts by importing them in much the same way as in SharePoint 2003. In my example, I'll use the `DeclarativeCatalogPart` to make the Web Parts available.

Once the `DeclarativeCatalogPart` is on the web page, you can use it to edit the underlying ASP.NET code. This is accomplished by editing the `WebPartsTemplate` property directly on the design surface. Figure 10-2 shows the control, which contains a blank text area used to enter the ASP.NET code that will declare a Web Part in the catalog.

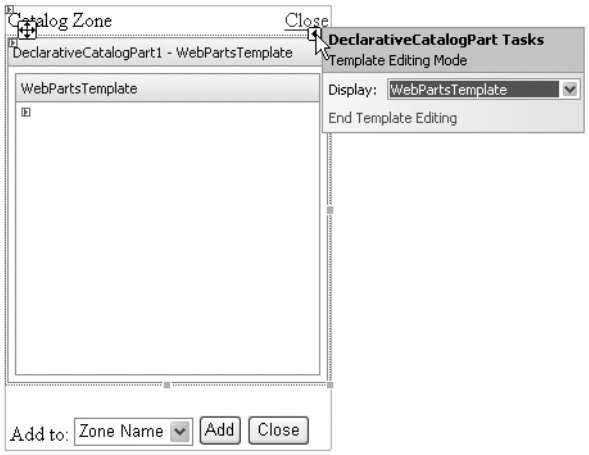

Figure 10-2. *A text area for modifying the WebPartsTemplate property*

While the WebPartsTemplate property is supposed to make it easy to add the necessary declarative code to the page, its behavior is incorrect in the current beta release. Therefore, I have found that you must edit the code directly in the Source view for the page. The only real challenge is figuring out where to place your code. I recommend attempting to make the change through the DeclarativeWebPart control first and then cleaning up the code in Source view.

Follow these steps to make the proper declaration:

1. Drag a CatalogZone control from the toolbox onto the design surface of the web page.
2. Drag a DeclarativeWebPart control from the toolbox into the CatalogZone control.
3. Click the Edit Templates hyperlink.
4. In the WebPartsTemplate text area, add a declaration in the form <tagprefix:➥ classname ID="id" Title="title" Runat="Server" />. The following code shows this declaration for the CViewer.Frame class I created earlier:

   ```
   <csharppart:Frame ID="mycspart" Title="C# Viewer" Runat="Server" />
   ```

5. Switch to Source view in the page and clean up the declaration as necessary to make it appear like the preceding code.

After the Web Parts are declared, they should be listed in the body of the DeclarativeWebPart in Design view. The only thing left to do is add a button to the page that will set the DisplayMode property of the WebPartManager control to display the catalog. Entering catalog mode is done with a single line of code similar to the following:

```
WebPartManager1.DisplayMode = WebPartManager.CatalogDisplayMode
```

Once in catalog mode, you can add any of the declared Web Parts to the zones defined by WebPartZone controls. When the Web Parts are added, they will show the images that were specified as the default values in code. Figure 10-3 shows the catalog with the Web Parts available for addition to a zone.

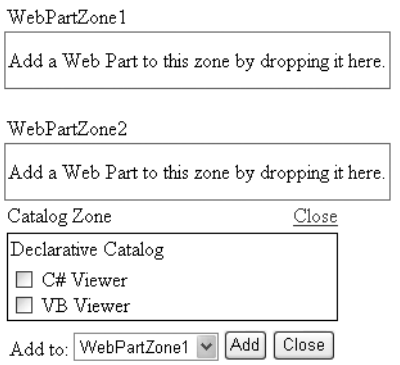

Figure 10-3. *Web Parts in the declarative catalog*

Personalizing Web Parts

At this point I can add Web Parts from the declarative catalog to the page, but I have no way to change the properties of the Web Parts. Both Web Parts simply display the default image specified in the class code. In order to make changes to the properties, I have to include some editing controls on the page and then decorate my properties with some special attributes.

Properties are edited using a combination of an `EditZone` control and a `PropertyGrid➥EditorPart` control. The `EditZone` control acts as a host for the `PropertyGridEditorPart` control, which creates the user interface necessary to edit Web Part properties. First you drag an `EditZone` control onto the page, and then you drag a `PropertyGridEditorPart` control on top of it. While you're at it, you can also drag an `AppearanceEditorPart` into the zone, which will allow you to edit basic properties such as the title of the Web Part. Figure 10-4 shows the current page, which I have cleaned up a bit through the use of an HTML table for formatting.

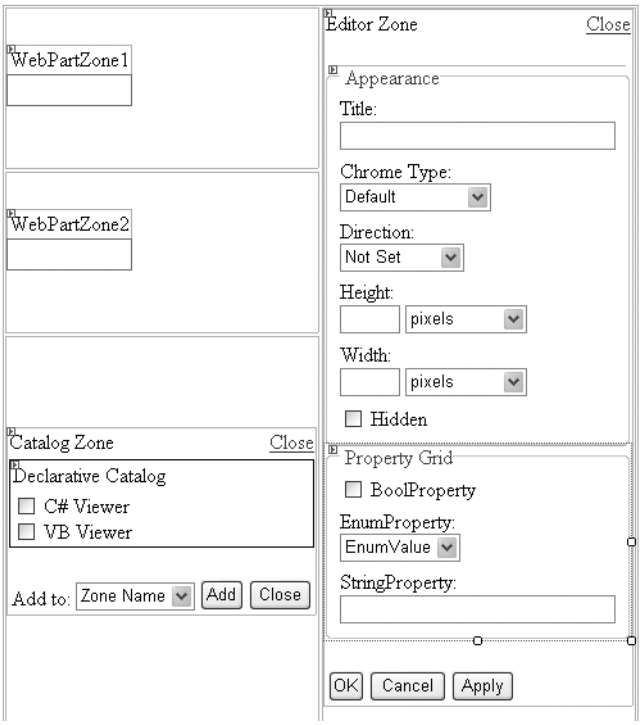

Figure 10-4. *A set of editing controls*

After the editing controls are on the page, you must add some attributes to the Web Part properties you wish to edit. By default, the properties that you create are hidden from the user unless you explicitly declare that they may be edited. This is exactly the same behavior as I discussed in Chapter 5 with SharePoint 2003 Web Parts; however, the attributes are different.

To expose a property for editing, you must decorate it with the `WebBrowsable` attribute. This attribute allows the `PropertyGridEditorPart` control to display the property value for editing in the page. Additionally, you can decorate the property with the `WebDisplayName` and

WebDescription attributes to show a property name and description respectively in the editor. Marking the property with the WebBrowsable attribute, however, will not save the changes to the page once the application is closed. If you want the changes to persist, then the property must also be decorated with the Personalizable attribute. List 10-5 shows the URL property decorated with the appropriate attributes in both C# and VB .NET.

Listing 10-5. *The URL Property*

```
//C# Property
[WebBrowsable(),WebDisplayName("URL"),
WebDescription("The URL of the image"),Personalizable()]public string Url
{
    get{return _url;}
    set{_url = value;}
}

'VB .NET Property
<WebBrowsable(), WebDisplayName("URL"), _
WebDescription("The URL of the image"), Personalizable()> _
Public Property URL() As String
    Get
        Return m_URL
    End Get
    Set(ByVal value As String)
        m_URL = value
    End Set
End Property
```

Whenever you create a new web site for use with Web Parts, Visual Studio automatically creates a SQL Server Express database to maintain personalized property values. You can see the database by opening the Server Explorer inside of VS2005. This database maintains the property values as set by each individual who is using the page.

The database associated with your web application remembers the property values for each user based on the security context with which they access the application. For applications that use Windows authentication, this happens automatically. However, you can also choose to use forms authentication in ASP.NET 2.0 to track the property assignments.

Once the editing environment is created and the properties are properly decorated, you can place the web page in edit mode. This is done by changing the DisplayMode property of the WebPartManager to EditDisplayMode. Once this is done, you may use the drop-down menu associated with any Web Part to change the property values. Figure 10-5 shows the final web page in edit mode.

Figure 10-5. *Editing Web Part properties*

Using Visual Studio Tools for Office

In Chapter 8, I showed the functionality of and discussed how to create several different solutions based on the Microsoft Office suite that were complementary to WSS. In some cases, the functionality was easy to incorporate, such as the Shared Workspace. However, in cases where you had to write custom code, such as for research applications and Smart Documents, the process was far from simple. Much of the custom coding in these types of applications is confusing and tedious, which may have discouraged you from trying to utilize them in your own organization. Fortunately, Microsoft has made some strides in solving these difficulties by shipping a new version of VSTO with VS2005 that makes application development with the Office suite much easier than it was before.

Understanding Project Types

When VSTO is installed with VS2005, the first thing you'll notice is that Microsoft Word and Excel project types are available directly from the New Project dialog. These project types allow you to create solutions that can utilize controls dragged directly from the toolbox onto a document or task pane. Additionally, you can build Smart Documents with a code-behind metaphor similar to any other project type. Figure 10-6 shows the New Project dialog with the VSTO project types displayed.

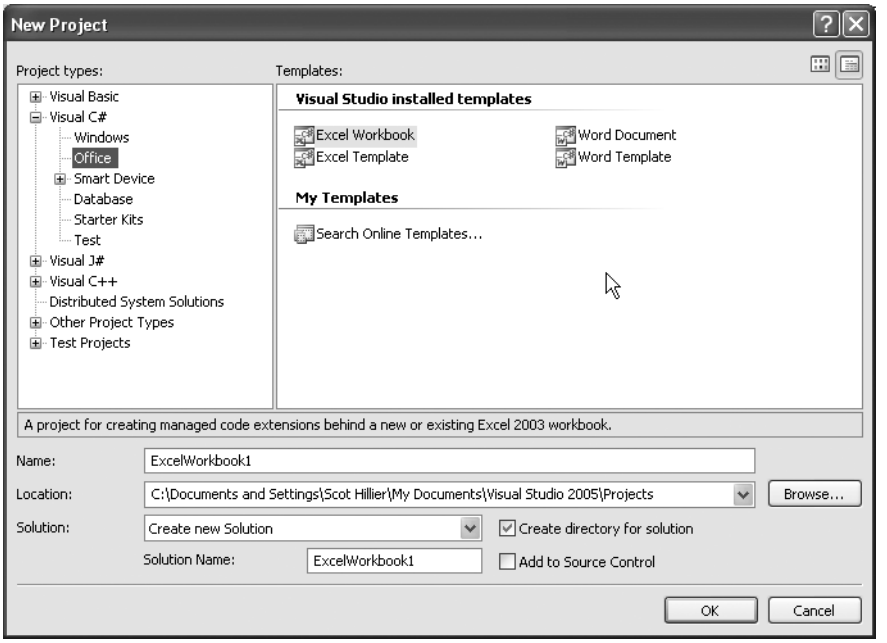

Figure 10-6. *VSTO project types*

When you select to create one of the new project types, VS2005 starts a project that acts as a host for either Word or Excel. This means that you can actually see the entire Word or Excel application running inside of VS2005. This capability allows you to treat an Office document like a Windows form. You can drag tools from the toolbox onto the document, double-click them, and write code behind the controls.

Follow these steps to add controls to a document:

1. Start Visual Studio 2005 and select File ➤ New ➤ Project from the main menu.
2. In the New Project dialog, expand the Visual Basic node and select the Office node from the Project Types tree.
3. In the Templates list, select the Word Template project.
4. Name the new project **HelloWord**.
5. Click the OK button to start the project wizard.
6. On the Select a Document for Your Application screen, choose to Create a New Document and click the OK button.
7. From the toolbox, drag a button onto the new Word document.
8. Double-click the button to open the code window.
9. In the `Click` event, add the following code:

```
MessageBox.Show("Hello, Word!")
```

10. Select Debug ➤ Start Without Debugging from the main menu.

11. When the application starts, click the button and display the message.

Along with placing controls directly on a document, you can also create your own interface on the task pane. I showed you how to do this in Chapter 8 using Visual Studio 2003, but VS2005 makes it much easier. All you have to do is programmatically add the controls when the document is loaded.

Follow these steps to add a control to the task pane:

1. Complete the preceding steps to create the HelloWord project.

2. In the Solution Explorer, select `ThisDocument.vb` and click the View Code button.

3. Inside the `ThisDocument` class, make the following declaration for a button:

```
Private WithEvents MyButton As New Button
```

4. In the code window, use the Class Name drop-down list to select the `MyButton` class.

5. Use the Method Name drop-down list to select the `Click` event.

6. Add the following code to the `Click` event to show a message:

```
MessageBox.Show("Hello, Task Pane!")
```

7. Add the following code to the `Startup` event of the `ThisDocument` class to load the button into the task pane:

```
MyButton.Text = "Push Me!"
ActionsPane.Controls.Add(MyButton)
```

8. Select Debug ➤ Start Without Debugging from the main menu.

9. When the application starts, click the button in the task pane and display the message.

Adding Smart Tags to Documents

In Chapter 8, I showed how to create Smart Tags for Office documents and trigger actions based on recognized terms. Building Smart Tags is another area that is complex in VS2003 but has been simplified in VS2005. Using VSTO, you can create Smart Tags in far fewer steps than before.

Smart Tags in VS2005 are associated with an Office document project like the ones created earlier in this chapter. To get started, you must set a reference to the Microsoft Smart Tags 2.0 Type Library in VS2005. Once the reference is set, you must add a new class to the project and inherit from `Microsoft.Office.Tools.Word.SmartTag`. As an example, I'll build Smart Tags that recognize the terms "DataLan", "Microsoft", and "Apress". The process begins by creating a class that references the appropriate namespaces. Listing 10-6 shows the starting code for the Smart Tags in both C# and VB .NET.

Listing 10-6. *Starting Smart Tag Code*

```
//C# Code
using System.Windows.Forms;
using Microsoft.Office.Tools.Word;
using Microsoft.Office.Interop.SmartTag;

namespace WordDocument1
{
    class companyTag : Microsoft.Office.Tools.Word.SmartTag
    {
    }
}

'VB .NET Code
Imports Microsoft.Office.Tools.Word
Imports Microsoft.Office.Interop.SmartTag

Public Class CompanyTag
    Inherits Microsoft.Office.Tools.Word.SmartTag
End Class
```

Next, you must override the class constructor and provide information about the terms to recognize as well as the actions to take upon recognition. This is a matter of declaring one or more `Action` variables and adding them to the `Actions` collection of the Smart Tag. Additionally, you must add the terms to recognize to the `Terms` collection of the Smart Tag. Listing 10-7 shows the overridden constructors for both C# and VB .NET.

Listing 10-7. *Overriding the Constructors*

```
//C# Code
Action nameAction;

public companyTag():base("www.datalan.com#sample", "Sample")
{
    nameAction = new Action("Show recognized name");
    Actions = new Action[] { nameAction };
    nameAction.Click +=new ActionClickEventHandler(nameAction_Click);
    Terms.Add("DataLan");
    Terms.Add("Microsoft");
    Terms.Add("Apress");
}

'VB .NET Code
Private WithEvents NameAction As Action
```

```
Sub New()
    MyBase.New("www.datalan.com#sample", "Sample")
    NameAction = New Action("Show recognized name")
    Actions = New Action() {nameAction}
    Terms.Add("Datalan")
    Terms.Add("Microsoft")
    Terms.Add("Apress")
End Sub
```

When a term from the `Terms` collection is recognized, a Smart Tag will appear in the document. Selecting the Smart Tag will display the associated items from the `Actions` collection. When a user selects one of the actions, your code runs the appropriate event; therefore, you must create an event for each of the actions. Listing 10-8 shows how to create an event in both C# and VB .NET.

Listing 10-8. *Capturing Action Events*

```
//C# Code
public void nameAction_Click(
    object sender,Microsoft.Office.Tools.Word.ActionEventArgs e)
{
    MessageBox.Show("The recognized name is: " + e.Range.Text);
}

'VB .NET Code
Private Sub MailAction_Click(ByVal sender As Object, _
ByVal e As Microsoft.Office.Tools.Word.ActionEventArgs) _
Handles NameAction.Click
    MessageBox.Show("The name is: " & e.Range.Text)
End Sub
```

Once the Smart Tag is complete, you must load it into memory when the document is opened. This is accomplished by adding it to the `VstoSmartTags` collection. Listing 10-9 shows how to add the Smart Tag in both C# and VB .NET.

Listing 10-9. *Adding a Smart Tag to a Document*

```
//C# Code

private void ThisDocument_Startup(object sender, System.EventArgs e)
{
    VstoSmartTags.Add(new companyTag());
}

'VB .NET Code
Private Sub ThisDocument_Startup(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles Me.Startup
    VstoSmartTags.Add(New CompanyTag)
End Sub
```

Exercise 10-1: Building a Web Part Page

With the release of ASP.NET 2.0, developers can now use Web Parts in custom applications outside of SharePoint 2003. Additionally, ASP.NET 2.0 is important because it will be the basis for the Web Parts framework in the next version of SharePoint. In this exercise, you will work with the Web Parts control set to create a simple application that uses Web Parts to display a business card.

Creating the New Project

Creating web site projects in VS2005 can be done directly against a web server or by using the local file system. These options give you some flexibility during development. In this project, you will create a new web application in C# using the local file system.

Follow these steps to create the new project:

1. Start Visual Studio 2005 and select File ➤ New ➤ Web Site from the main menu.
2. In the New Web Site dialog, select the ASP.NET Web Site template.
3. In the Location drop-down list, select File System.
4. In the Language drop-down list, select Visual C#.
5. Click the Browse button.
6. In the Choose Location dialog, select a location in the file system tree to create the new web site.
7. Create a new folder and name it **BusinessCard**.
8. Click the Open button to return to the New Web Site dialog.
9. In the New Web Site dialog, click the OK button to create the new web site.

Adding Site Membership

Implementing site membership allows you to control access to your web site projects. Additionally, site membership includes the capability for users to customize Web Parts individually and have those customizations persist across sessions. In this section, you will work with VS2005 to set up membership restrictions on your web site.

Follow these steps to set up membership rules:

1. In the Solution Explorer, select the project and click the ASP.NET Configuration button.
2. In the ASP.NET Web Site Administration Tool, click the Security hyperlink.
3. On the Security tab, click the link entitled Use the Security Setup Wizard to Configure Security Step by Step.
4. On the Welcome page, click the Next button.

5. On the Select Access Method page, choose From a Local Area Network and click the Next button.
6. On the Add New Access Rules page, select to add a user and type the name of your account.
7. Set the permission option to Allow and click the button labeled Add This Rule.
8. Click the Finish button.
9. Close the ASP.NET Web Site Administration Tool.

Creating the Logo Web Part

Web Parts can be created in a variety of ways in VS2005. You can use standard controls as Web Parts or create custom controls. In the next few sections, you will create Web Parts for the application using user controls. The first control you will create is for displaying a logo on the business card.

Follow these steps to create the Web Part:

1. In the Solution Explorer, right-click the project and select Add New Item from the context menu.
2. In the Add New Item dialog, select to add a new web user control.
3. Name the new web user control **LogoPart.ascx** and click the Add button.
4. In the Solution Explorer, select the `LogoPart.ascx` file and click the View Designer button.
5. In the toolbox, expand the Standard control section and drag an `Image` control from the toolbox onto the designer surface.
6. Select the `Image` control on the designer surface to display the Properties window.
7. In the Properties window, change the AlternateText property to read **Insert Logo**.
8. In the Solution Explorer, select the `LogoPart.ascx` file and click the View Code button.
9. Add the following code to define a new property for the control:

```
[WebBrowsable(),WebDisplayName("LogoURL"),
WebDescription("The URL for the logo image.")]
public string logoURL
{
    get{return Image1.ImageUrl;}
    set{Image1.ImageUrl = value;}
}
```

Creating the Text Web Part

The next Web Part you will create is for displaying simple text on the card. This Web Part will be used in several places on the web page.

Follow these steps to create the Web Part:

1. In the Solution Explorer, right-click the project and select Add New Item from the context menu.
2. In the Add New Item dialog, select to add a new web user control.
3. Name the new web user control **TextPart.ascx** and click the Add button.
4. In the Solution Explorer, select the `TextPart.ascx` file and click the View Designer button.
5. In the toolbox, expand the Standard control section and drag a `Label` control from the toolbox onto the designer surface.
6. Select the `Label` control on the designer surface to display the Properties window.
7. In the Properties window, change the Text property to read **Insert Text**.
8. In the Solution Explorer, select the `TextPart.ascx` file and click the View Code button.
9. Add the following code to define a new property for the control:

```
[WebBrowsable(), WebDisplayName("CardText"),
WebDescription("The text to display.")]
public string cardText
{
    get { return Label1.Text; }
    set { Label1.Text = value; }
}
```

Creating the Link Web Part

Because the business card is based on a web page, you can have some extra built-in functionality. In this case, you will create a Web Part that supports hyperlink navigation. You will use this Web Part to display the e-mail address for the card so viewers can send mail directly by clicking the link.

Follow these steps to build the Web Part:

1. In the Solution Explorer, right-click the project and select Add New Item from the context menu.
2. In the Add New Item dialog, select to add a new web user control.
3. Name the new web user control **LinkPart.ascx** and click the Add button.
4. In the Solution Explorer, select the `LinkPart.ascx` file and click the View Designer button.

5. In the toolbox, expand the Standard control section and drag a `HyperLink` control from the toolbox onto the designer surface.
6. Select the `HyperLink` control on the designer surface to display the Properties window.
7. In the Properties window, change the Text property to read **Insert Text**.
8. In the Solution Explorer, select the `LinkPart.ascx` file and click the View Code button.
9. Add the following code to define new properties for the control:

```
[WebBrowsable(), WebDisplayName("LinkText"),
WebDescription("The text of the link.")]
public string linkText
{
    get { return HyperLink1.Text; }
    set { HyperLink1.Text = value; }
}

[WebBrowsable(), WebDisplayName("LinkURL"),
WebDescription("The URL for the link.")]
public string linkUrl
{
    get { return HyperLink1.NavigateUrl; }
    set { HyperLink1.NavigateUrl = value; }
}
```

Creating the Business Card

Once the Web Parts are complete, you are ready to create the business card. The business card is created by defining a layout using a table element and filling the table with zones. After the zones are defined, you will place Web Parts in them to complete the user interface.

Follow these steps to create the business card:

1. In the Solution Explorer, select `Default.aspx` and click the View Designer button.
2. In the Toolbox, expand the WebParts control section and drag a `WebPartManager` control from the toolbox onto the designer surface.
3. Select Layout ➤ Insert Table from the main menu in Visual Studio.
4. In the Insert Table dialog, set the number of rows to **4** and the number of columns to **2**.
5. Click the OK button to insert the new table.
6. Select the upper two cells in the left-hand column. Then right-click the cells and select Merge Cells from the context menu.
7. Select the lower two cells in the left-hand column. Then right-click the cells and select Merge Cells from the context menu. Your table should now appear as shown in Figure 10-7.

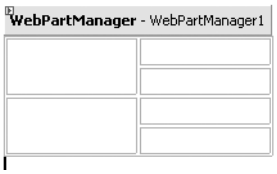

Figure 10-7. *The formatted table*

8. Drag a `WebPartZone` control from the toolbox into the upper-left hand cell in the table.
9. When you drop the `WebPartZone` control, a task window should appear. Select AutoFormat from this task window.
10. In the AutoFormat dialog, select Classic from the Scheme list and click the OK button to apply the format.
11. Repeat this process until you have placed a `WebPartZone` control in each cell. Your page should now appear as shown in Figure 10-8.

Figure 10-8. *Adding Web Part Zones to the table*

12. After the Web Part zones are created, drag the `LogoPart` web user control from the Solution Explorer into the upper left-hand zone.
13. Drag the `LinkPart` web user control into the lower right-hand zone.
14. Drag the `TextPart` web user control into each of the remaining zones. Your page should now appear as shown in Figure 10-9.

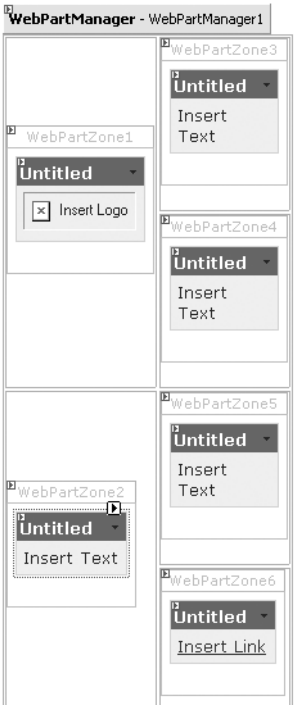

Figure 10-9. *The completed web page*

15. Drag an `EditorZone` control from the toolbox and drop it below the completed table.

16. Drag a `PropertyGridEditorPart` control from the toolbox and drop it inside the `EditorZone` control.

17. Drag an `AppearanceEditorPart` control from the toolbox and drop it inside the `EditorZone` control. The combination of the `EditorZone`, `PropertyGridEditorZone`, and `AppearanceEditorPart` controls will allow you to change the properties of the Web Parts.

18. Expand the Standard control section, drag a `Button` control from the toolbox, and drop it underneath the `EditorZone` control.

19. Using the Properties window, change the Text property of the button to **Design View**. Figure 10-10 shows the control set for editing properties.

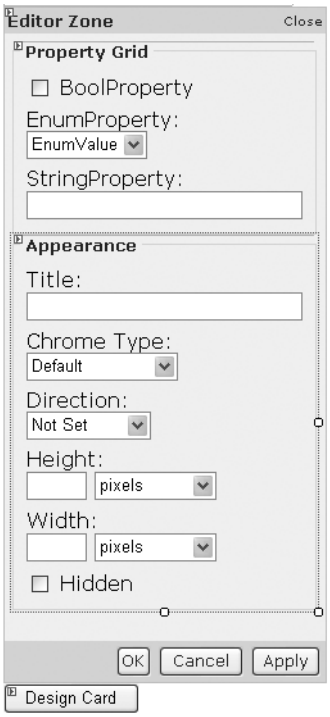

Figure 10-10. *The completed editing control set*

20. In the Solution Explorer, select `Default.aspx` and click the View Code button.

21. Add the following code to switch the card into design mode:

```
protected void Button1_Click(object sender, EventArgs e)
{
    if(Button1.Text=="Design Card")
    {
            WebPartManager1.DisplayMode = WebPartManager.EditDisplayMode;
            Button1.Text="View Card";
        }
        else
        {
            WebPartManager1.DisplayMode = WebPartManager.BrowseDisplayMode;
            Button1.Text="Design Card";
        }
    }
```

Testing the Solution

Once the business card is created, you can test the solution. You should be able to easily change the properties and save them. You also should be able to stop and start the application and see that the changes persist.

Follow these steps to test the application:

1. Select Debug ➤ Start Without Debugging from the main menu in Visual Studio.
2. When the application starts, you should see the business card with the default entries contained in each zone.
3. Click the Design Card button.
4. When the page enters design mode, click the down arrow associated with each zone and select Edit from the menu.
5. In the Property Grid, change the properties of the Web Parts to appropriate values for your business card.
6. In the Appearance pane, select None from the Chrome Type drop-down list for each Web Part. Figure 10-11 shows an example card created using the application.

Figure 10-11. *The final card*

Exercise 10-2: Building a Smart Document

Building Smart Documents in previous versions of Visual Studio could be tedious and difficult. The Visual Studio Tools for Office (VSTO) found in VS2005 make development of such documents much easier. In this exercise, you will build a smart invoice document using VSTO. This Smart Document is similar to the one you created in Chapter 8.

Setting Up the Project

When VSTO is installed with VS2005, you can create project types based on Word or Excel. Furthermore, Word and Excel are hosted directly in the development environment. In this exercise, you will create a project based on a Word document.

Follow these steps to start the project:

1. Start Visual Studio 2005 and select File ➤ New ➤ Project from the main menu.
2. In the New Project dialog, expand the Visual Basic node and select the Office node from the Project Types tree.

3. In the Templates list, select the Word Template project.
4. Name the new project **SmartInvoice2005**.
5. Click the OK button to start the project wizard.
6. On the Select a Document for Your Application screen, choose to Create a New Document and click the OK button.

Creating the XML Schema

The invoice we are creating uses an XML schema to define the elements in the invoice. Defining the elements in a Word document with XML makes it easier to detect when items of interest are added to the document. In this project, you will be interested in product names, identifiers, quantity, and price.

Follow these steps to define the XML schema:

1. In Visual Studio, select Project ➤ Add New Item from the main menu.
2. In the Add New Item dialog, select XML Schema and name the file **SmartInvoice2005.xsd**.
3. After the new schema is created, right-click the schema design surface and select Add ➤ New Element from the context menu.
4. Name the new element **ProductID** and change its type to string.
5. Add a new element named **ProductName** and change its type to string.
6. Add a new element named **QuantityPerUnit** and change its type to string.
7. Add a new element named **UnitPrice** and change its type to double. Figure 10-12 shows the completed schema in the Visual Studio designer.

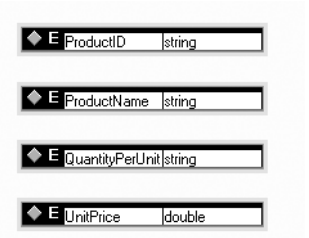

Figure 10-12. *The Smart Document schema*

8. In the Solution Explorer, double-click the file `ThisDocument.vb` to open the document template in Visual Studio.
9. Select Tools ➤ Microsoft Office Word Tools ➤ Templates and Add-Ins.
10. In the Templates and Add-Ins dialog, click the XML Schema tab.
11. On the XML Schema tab, click the Add Schema button.

12. In the Add Schema dialog, navigate to your project directory and double-click the file `SmartInvoice2005.xsd`.
13. In the Schema Settings dialog, uncheck the box entitled Changes Affect Current User Only.
14. In the Alias field, type **SmartInvoice2005** and click the OK button.
15. In the Templates and Add-Ins dialog, click the OK button to finish adding the new schema to the document template.

Creating the Invoice Template

Once the XML schema is defined for the document, you can use it to define fields. In this project, you will define a table where line items for an invoice can be entered. You will go on to define this document as a template so that it can be reused.

Follow these steps to define the document template:

1. With the document template open, select Table ➤ Insert ➤ Table from the main menu.
2. In the Insert Table dialog, set the number of columns to **4** and the number of rows to **6**.
3. Click the OK button to add the table to the document template.
4. With the cursor inside the new table, select Table ➤ Table AutoFormat from the main menu.
5. In the Table AutoFormat dialog, select the Table Classic 1 format and click the Apply button.
6. Name the table columns **Number**, **Name**, **Quantity**, and **Price** from left to right.
7. Using the cursor, select the four table cells immediately under the Name column.
8. With the cells selected, click the ProductID element that appears in the XML Structure pane.
9. When prompted, select to apply the element to the current selection. This will place XML nodes in the table column.
10. Repeat steps 7 through 9 to add the remaining XML nodes to the appropriate column. Figure 10-13 shows the completed table in Visual Studio.

Number	*Name*	*Quantity*	**Price**
ProductID ProductID	ProductName ProductName	QuantityPerUnit QuantityPerUnit	UnitPrice UnitPrice
ProductID ProductID	ProductName ProductName	QuantityPerUnit QuantityPerUnit	UnitPrice UnitPrice
ProductID ProductID	ProductName ProductName	QuantityPerUnit QuantityPerUnit	UnitPrice UnitPrice
ProductID ProductID	ProductName ProductName	QuantityPerUnit QuantityPerUnit	UnitPrice UnitPrice

Figure 10-13. *Applying the XML to the table*

Coding the Solution

Coding the Smart Document involves creating a user interface, handling user actions, and managing the document data. You begin coding the solution by setting up references to key namespaces in the project. In the Solution Explorer, select the file `ThisDocument.vb` and click the View Code button. Then add the following code to the project:

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports Word = Microsoft.Office.Interop.Word

Public Class ThisDocument
End Class
```

Adding Controls

The user interface associated with the Smart Document is a set of controls residing in the task pane. Based on the location of the cursor within the table of invoice items, the controls in the task pane will adjust to allow information to be inserted into the document. The product data that appears inside the task pane is managed in a `DataSet` object. The `DataSet` object is decorated with the `Cached` attribute, which allows the document to be taken offline and still work because the data is cached within the document itself. Add the following code to the class to declare the necessary variables to create the user interface and manage the underlying data:

```
Private WithEvents lstProductID As New ListBox
Private txtName As New TextBox
Private txtQuantity As New TextBox
Private txtPrice As New TextBox
Private WithEvents btnInsertID As New Button
Private WithEvents btnInsertName As New Button
Private WithEvents btnInsertQuantity As New Button
Private WithEvents btnInsertPrice As New Button

'Cached DataSet
<Cached()> Private objProductSet As DataSet

'Current Selection
Private objSelection As Word.Selection
```

Initializing the Data

When the document is first loaded, it must check to see whether the `DataSet` is populated. If the document has been previously used, then the `DataSet` will be filled from cached data. If the `DataSet` is empty, however, it must be populated from the database. In this exercise, you are simply using the Northwind database commonly available in SQL Server. Add the code from Listing 10-10 to the document class to initialize the data. Be sure to modify the connection string to reflect your environment.

Listing 10-10. *Initializing the Data*

```
Private Sub ThisDocument_Startup(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles Me.Startup

    'Add Controls to Actions Pane
    With ActionsPane.Controls
        .Add(lstProductID)
        btnInsertID.Text = "Insert Number"
        btnInsertID.Enabled = False
        .Add(btnInsertID)
        .Add(txtName)
        btnInsertName.Text = "Insert Name"
        btnInsertName.Enabled = False
        .Add(btnInsertName)
        .Add(txtQuantity)
        btnInsertQuantity.Text = "Insert Quantity"
        btnInsertQuantity.Enabled = False
        .Add(btnInsertQuantity)
        .Add(txtPrice)
        btnInsertPrice.Text = "Insert Price"
        btnInsertPrice.Enabled = False
        .Add(btnInsertPrice)
    End With

    'Get Product Information if not already cached
    If objProductSet Is Nothing Then

        Try

            'Connection string
            Dim strConnection As String = {replace with your string}

            'SQL string
            Dim strSQL As String = "Select ProductID, ProductName,
            QuantityPerUnit, UnitPrice FROM Products"

            'Run query
            With New SqlDataAdapter
                objProductSet = New DataSet("root")
                .SelectCommand = New SqlCommand(strSQL, _
                New SqlConnection(strConnection))
                .Fill(objProductSet, "Products")
            End With

        Catch x As Exception
            MsgBox(x.Message)
        End Try

    End If
```

```
    'Fill List with Product IDs
    With lstProductID
        .DataSource = objProductSet.Tables("Products")
        .DisplayMember = "ProductName"
        .ValueMember = "ProductID"
    End With

End Sub
```

Managing the User Interface

As the user interacts with the controls in the task pane, they must be updated to reflect the products selected or actions taken. Changes in the status of the control set are also made based on the current location of the cursor within the XML nodes on the document. Add the code from Listing 10-11 to create the required behaviors.

Listing 10-11. *Managing the Control Set*

```
Private Sub lstProductID_SelectedIndexChanged(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles lstProductID.SelectedIndexChanged

    'Update the text boxes
    txtName.Text = DirectCast(lstProductID.SelectedItem, _
    DataRowView).Row("ProductName").ToString
    txtQuantity.Text = DirectCast(lstProductID.SelectedItem, _
    DataRowView).Row("QuantityPerUnit").ToString
    txtPrice.Text = DirectCast(lstProductID.SelectedItem, _
    DataRowView).Row("UnitPrice").ToString

End Sub

Private Sub ProductIDNode_ContextEnter(ByVal sender As Object, _
ByVal e As Microsoft.Office.Tools.Word.ContextChangeEventArgs) Handles _
ProductIDNode.ContextEnter
    btnInsertID.Enabled = True
End Sub

Private Sub ProductIDNode_ContextLeave(ByVal sender As Object, _
ByVal e As Microsoft.Office.Tools.Word.ContextChangeEventArgs) Handles _
ProductIDNode.ContextLeave
    btnInsertID.Enabled = False
End Sub

Private Sub ProductNameNode_ContextEnter(ByVal sender As Object, _
ByVal e As Microsoft.Office.Tools.Word.ContextChangeEventArgs) Handles _
ProductNameNode.ContextEnter
    btnInsertName.Enabled = True
End Sub
```

```
Private Sub ProductNameNode_ContextLeave(ByVal sender As Object, _
ByVal e As Microsoft.Office.Tools.Word.ContextChangeEventArgs) Handles _
ProductNameNode.ContextLeave
    btnInsertName.Enabled = False
End Sub

Private Sub QuantityPerUnitNode_ContextEnter(ByVal sender As Object, _
ByVal e As Microsoft.Office.Tools.Word.ContextChangeEventArgs) Handles _
QuantityPerUnitNode.ContextEnter
    btnInsertQuantity.Enabled = True
End Sub

Private Sub QuantityPerUnitNode_ContextLeave(ByVal sender As Object, _
ByVal e As Microsoft.Office.Tools.Word.ContextChangeEventArgs) Handles _
QuantityPerUnitNode.ContextLeave
    btnInsertQuantity.Enabled = False
End Sub

Private Sub UnitPriceNode_ContextEnter(ByVal sender As Object, _
ByVal e As Microsoft.Office.Tools.Word.ContextChangeEventArgs) Handles _
UnitPriceNode.ContextEnter
    btnInsertPrice.Enabled = True
End Sub

Private Sub UnitPriceNode_ContextLeave(ByVal sender As Object, _
ByVal e As Microsoft.Office.Tools.Word.ContextChangeEventArgs) Handles _
UnitPriceNode.ContextLeave
    btnInsertPrice.Enabled = False
End Sub
```

Inserting Text into the Document

When a user has selected an appropriate column in the document and the controls are updated, they can insert text into the document. Inserting text is done by capturing the current selection within the document. The selection indicates the current cursor location and therefore where the text should be inserted. Add the code from Listing 10-12 to complete the project.

Listing 10-12. *Inserting Text*

```
Private Sub ThisDocument_SelectionChange(ByVal sender As Object, _
ByVal e As Microsoft.Office.Tools.Word.SelectionEventArgs) Handles _
Me.SelectionChange
    objSelection = e.Selection
End Sub

Private Sub btnInsertID_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles btnInsertID.Click
    objSelection.Text = DirectCast(lstProductID.SelectedItem, _
    DataRowView).Row("ProductID").ToString
End Sub
```

```
Private Sub btnInsertName_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles btnInsertName.Click
    objSelection.Text = DirectCast(lstProductID.SelectedItem, _
    DataRowView).Row("ProductName").ToString
End Sub

Private Sub btnInsertQuantity_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles btnInsertQuantity.Click
    objSelection.Text = DirectCast(lstProductID.SelectedItem, _
    DataRowView).Row("QuantityPerUnit").ToString
End Sub

Private Sub btnInsertPrice_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles btnInsertPrice.Click
    objSelection.Text = DirectCast(lstProductID.SelectedItem, _
    DataRowView).Row("UnitPrice").ToString
End Sub
```

Testing the Solution

Once the coding is complete, you can run the project by selecting Debug ➤ Start Without Debugging from the main menu. When the application starts, you should see the table of items in the invoice. Start by selecting a product in the product list located in the task pane. Next, place your cursor in various columns of the table and check that the appropriate insert button enables. Click the button and try inserting some text. Figure 10-14 shows the completed user interface in the task pane.

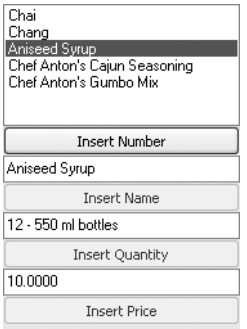

Figure 10-14. *The Smart Document task pane*