PART VIII

Additional Topics

www.it-ebooks.info

www.it-ebooks.info

CHAPTER 30

Interacting with Windows Forms

In an ideal world, once developers master a new technology such as WPF, they'd leave the previous framework behind. Everything would be written using the latest, most capable toolkit, and no one would ever worry about legacy code. Of course, this ideal world is nothing like the real world, and most WPF developers will need to interact with the Windows Forms platform at some point for two reasons: to leverage existing code investments and to compensate for missing features in WPF.

In this chapter, you'll look at strategies for integrating Windows Forms and WPF content. You'll consider how to use both types of windows in a single application, and you'll explore the more impressive trick of mixing content from both platforms in a single window. But before you delve into WPF and Windows Forms interoperability, it's worth taking a step back and assessing the reasons you should (and shouldn't) use WPF interoperability.

What's New Early betas of WPF 4.5 included a mechanism that solved the airspace issue (the inability to overlap content that's created by WPF with content that's created by Windows Forms). However, this feature was dropped from the final release, leaving WPF's interoperability support unchanged.

Assessing Interoperability

There's no tool to transform Windows Forms interfaces into similar WPF interfaces (and even if there were, such a tool would be only a starting point of a long and involved migration process). Of course, there's no *need* to transplant a Windows Forms application into the WPF environment—most of the time, you're better off keeping old applications as is and using WPF for new projects. However, life isn't always that simple. You might decide that you want to add a WPF feature (such as an eye-catching 3-D animation) to an existing Windows Forms application. Or you might decide that you want to migrate an existing Windows Forms application to WPF piece by piece, as you release updated versions. Either way, the interoperability support in WPF can help you make the transition gradually and without sacrificing the work that's invested in your legacy code.

Before you toss WPF elements and Windows Forms controls together, it's important to assess your overall goals. In many situations, developers are faced with deciding between incrementally enhancing a Windows Forms application (and gradually moving it into the WPF world) or replacing it with a newly rewritten WPF masterpiece. Obviously, the first approach is faster and easier to test, debug, and release.

www.it-ebooks.info

However, in a suitably complex application that needs a major WPF injection, there may come a point where it's simpler to start over in WPF and import the legacy bits that you need.

■ **Note** As always, when moving from one user interface platform to another, you should be forced to migrate only the user interface. Other details, such as data access code, validation rules, file access, and so on, should be abstracted away in separate classes (and possibly even separate assemblies), which you can plug into a WPF front-end just as easily as a Windows Forms application. Of course, this level of componentization isn't always possible, and sometimes other details (such as data-binding considerations and validation strategies) can lead you to shape your classes a certain way and inadvertently limit their reusability.

Mixing Windows and Forms

The cleanest way to integrate WPF and Windows Forms content is to place each in a separate window. That way, your application consists of well-encapsulated window classes, each of which deals with just a single technology. Any interoperability details are handled in the *glue* code—the logic that creates and shows your windows.

Adding Forms to a WPF Application

The easiest approach to mixing windows and forms is to add one or more forms (from the Windows Forms toolkit) to an otherwise ordinary WPF application. Visual Studio makes this easy—just right-click the project name in the Solution Explorer and choose Add ➤ New Item. Then select the Windows Forms category on the left side and choose the Windows Form template. Finally, give your form a file name and click Add. The first time you add a form, Visual Studio adds references to all the required Windows Forms assemblies, including System.Windows.Forms.dll and System.Drawing.dll.

You can design a form in a WPF project in the same way that you design it in a Windows Forms project. When you open a form, Visual Studio loads the normal Windows Forms designer and fills the Toolbox with Windows Forms controls. When you open the XAML file for a WPF window, you get the familiar WPF design surface instead.

■ **Note** For better separation between WPF and Windows Forms content, you might choose to place the "foreign" content in a separate class library assembly. For example, a Windows Forms application might use the WPF windows defined in a separate assembly. This approach makes especially good sense if you plan to reuse some of these windows in both Windows Forms and WPF applications.

Adding WPF Windows to a Windows Forms Application

The reverse trick is a bit more awkward. Visual Studio doesn't directly allow you to create a new WPF window in a Windows Forms application. (In other words, you won't see it as one of the available templates when you right-click your project and choose Add New Item.) However, you can add the existing .cs and .xaml files that define a WPF window from another WPF project. To do so, right-click your project in the

www.it-ebooks.info

Solution Explorer, choose Add ➤ Existing Item, and find both these files. You'll also need to add references to the core WPF assemblies (PresentationCore.dll, PresentationFramework.dll, and WindowsBase.dll).

■ **Tip** There's a shortcut to adding the WPF references you need. You can add a WPF user control (which Visual Studio *does* support), which causes Visual Studio to add these references automatically. You can then delete the user control from your project. To add a WPF user control, right-click the project, choose Add ➤ New Item, pick the WPF category, and select the User Control (WPF) template.

After you add a WPF window to a Windows Forms application, that window is treated correctly. When you open it, you'll be able to use the WPF designer to modify it. When you build the project, the XAML will be compiled, and the automatically generated code will be merged with your code-behind class, just as it is in a full-fledged WPF application.

Creating a project that uses forms and windows isn't too difficult. However, there are a few extra considerations when you show these forms and windows at runtime. If you need to show a window or form modally (as you would with a dialog box), the task is straightforward, and your code is essentially unchanged. But if you want to show a window modelessly, you need a bit of extra code to ensure proper keyboard support, as you'll see in the following sections.

Showing Modal Windows and Forms

Showing a modal form from a WPF application is effortless. You use exactly the same code you'd use in a Windows Forms project. For example, if you have a form class named Form1, you'd use code like this to show it modally:

```
Form1 frm = new Form1();
if (frm.ShowDialog() == System.Windows.Forms.DialogResult.OK)
{
    MessageBox.Show("You clicked OK in a Windows Forms form.");
}
```

You'll notice that the Form.ShowDialog() method works in a slightly different way than WPF's Window.ShowDialog() method. While Window.ShowDialog() returns true, false, or null, Form.ShowDialog() returns a value from the DialogResult enumeration.

The reverse trick—showing a WPF window from a form—is just as easy. Once again, you simply interact with the public interface of your Window class, and WPF takes care of the rest:

```
Window1 win = new Window1();
if (win.ShowDialog() == true)
{
    MessageBox.Show("You clicked OK in a WPF window.");
}
```

Showing Modeless Windows and Forms

It's not quite as straightforward if you want to show windows or forms modelessly. The challenge is that keyboard input is received by the root application and needs to be delivered to the appropriate window. In order for this to work between WPF and Windows Forms content, you need a way to forward these messages along to the right window or form.

www.it-ebooks.info

If you want to show a WPF window modelessly from inside a Windows Forms application, you must use the static ElementHost.EnableModelessKeyboardInterop() method. You'll also need a reference to the WindowsFormsIntegration.dll assembly, which defines the ElementHost class in the System.Windows.Forms.Integration namespace. (You'll learn more about the ElementHost class later in this chapter.)

You call the EnableModelessKeyboardInterop() method after you create the window but before you show it. When you call it, you pass in a reference to the new WPF window, as shown here:

```
Window1 win = new Window1();
ElementHost.EnableModelessKeyboardInterop(win);
win.Show();
```

When you call EnableModelessKeyboardInterop(), the ElementHost adds a message filter to the Windows Forms application. This message filter intercepts keyboard messages when your WPF window is active and forwards them to your window. Without this detail, your WPF controls won't receive any keyboard input.

If you need to show a modeless Windows Forms application inside a WPF application, you use the similar WindowsFormsHost.EnableWindowsFormsInterop() method. However, you don't need to pass in a reference to the form you plan to show. Instead, you simply need to call this method once before you show any form. (One good choice is to call this method at application startup.)

```
WindowsFormsHost.EnableWindowsFormsInterop();
```

Now you can show your form modelessly without a hitch:

```
Form1 frm = new Form1();
frm.Show();
```

Without the call to EnableWindowsFormsInterop(), your form will still appear, but it won't recognize all keyboard input. For example, you won't be able to use the Tab key to move from one control to the next.

You can extend this process to multiple levels. For example, you could create a WPF window that shows a form (modally or modelessly), and that form could then show a WPF window. Although you won't need to do this very often, it's more powerful than the element-based interoperability support you'll learn about later. This support allows you to integrate different types of content in the same window but doesn't allow you to nest more than one layer deep (for example, creating a WPF window that contains a Windows Forms control that, in turn, hosts a WPF control).

Enabling Visual Styles for Windows Forms Controls

When you show a form in a WPF application, that form uses the shockingly old-fashioned (pre–Windows XP) styles for buttons and other common controls. That's because support for the newer styles must be explicitly enabled by calling the Application.EnableVisualStyles() method. Ordinarily, Visual Studio adds this line of code to the Main() method of every new Windows Forms application. However, when you create a WPF application, this detail isn't included.

To resolve this issue, just call the EnableVisualStyles() method once before showing any Windows Forms content. A good place to do this is at the start of the application, as shown here:

```
public partial class App : System.Windows.Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        // Raises the Startup event.
        base.OnStartup(e);
```

www.it-ebooks.info

```
        System.Windows.Forms.Application.EnableVisualStyles();
    }
}
```

Notice that the EnableVisualStyles() method is defined in the System.Windows.Forms.Application class, *not* the System.Windows.Application class that forms the core of your WPF application.

Creating Windows with Mixed Content

In some cases, the clean window-by-window separation isn't suitable. For example, you might want to place WPF content in an existing form alongside Windows Forms content. Although this model is conceptually messier, WPF handles it quite gracefully.

In fact, including Windows Forms content in a WPF application (or vice versa) is more straightforward than adding ActiveX content to a Windows Forms application. In the latter scenario, Visual Studio must generate a wrapper class that sits between the ActiveX control and your code, which manages the transition from managed to unmanaged code. This wrapper is *component-specific*, which means each ActiveX control you use requires a separate customized wrapper. And because of the quirks of COM, the interface exposed by the wrapper might not match the interface of the underlying component exactly.

When integrating Windows Forms and WPF content, you don't need a wrapper class. Instead, you use one of a small set of containers, depending on the scenario. These containers work with any class, so there's no code-generation step. This simpler model is possible because even though Windows Forms and WPF are dramatically different technologies, they are both firmly grounded in the world of managed code.

The most significant advantage of this design is that you can interact with Windows Forms controls and WPF elements in your code directly. The interoperability layer comes into effect only when this content is rendered in the window. This part takes place automatically without requiring any developer intervention. You also don't need to worry about keyboard handling in modeless windows because the interoperability classes you'll use (ElementHost and WindowsFormsHost) handle that automatically.

Understanding WPF and Windows Forms "Airspace"

To integrate WPF and Windows Forms content in the same window, you need to be able to segregate a portion of your window for "foreign" content. For example, it's completely reasonable to throw a 3-D graphic into a Windows Forms application because you can place that 3-D graphic in a distinct region of a window (or even make it take up the entire window). However, it's not easy or worthwhile to reskin all the buttons in your Windows Forms application by making them WPF elements, because you'll need to create a separate WPF region for each button.

Along with the considerations of complexity, some things just aren't possible with WPF interoperability. For example, you can't *combine* WPF and Windows Forms content by overlapping it. That means you can't have a WPF animation send an element flying over a region that's rendered with Windows Forms. Similarly, you can't overlap partially transparent Windows Forms content over a WPF region to blend them together. Both of these violate what's known as the *airspace rule*, which dictates that WPF and Windows Forms must always have their own distinct window regions, which they manage exclusively. Figure 30-1 shows what's allowed and what isn't.

www.it-ebooks.info

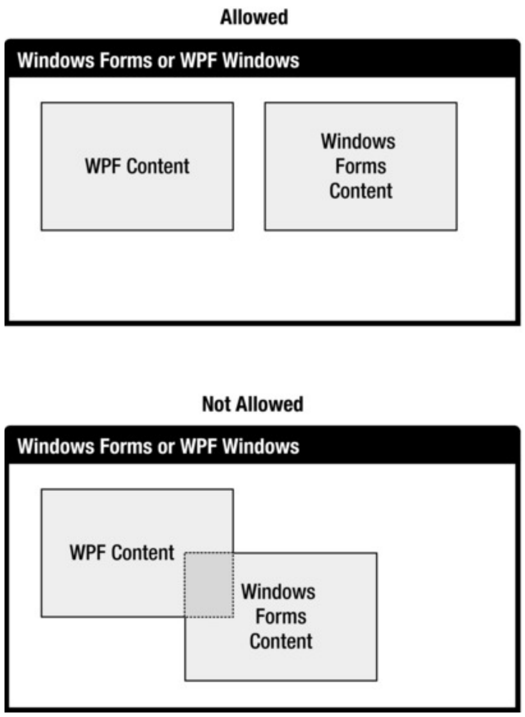


***Figure 30-1.** The airspace rule*

Technically, the airspace rule results from the fact that in a window containing WPF content and Windows Forms content, both regions have a separate window handle, or *hwnd*. Each hwnd is managed, rendered, and refreshed separately.

Window handles are managed by the Windows operating system. In classic Windows applications, every control is a separate window, which means each control has ownership of a distinct piece of screen real estate. Obviously, this type of "window" isn't the same as the top-level windows that float around your screen—it's simply a self-contained region (rectangular or otherwise). In WPF, the model is dramatically different—there's a single, top-level hwnd, and the WPF engine does the compositing for the entire window, which allows more-pleasing rendering (for example, effects such as dynamic anti-aliasing) and far greater flexibility (for example, visuals that render content outside their bounds).

■ **Note** A few WPF elements use separate window handles. These include menus, tooltips, and the drop-down portion of a combo box, all of which need the ability to extend beyond the bounds of the window.

The implementation of the airspace rule is fairly straightforward. If you place Windows Forms content on top of WPF content, you'll find that the Windows Forms content is always on top, no matter where it's declared in the markup or what layout container you use. That's because the WPF content is a single window, and the container with Windows Forms content is implemented as a separate window that's displayed on top of a portion of the WPF window.

If you place WPF content in a Windows Forms form, the result is a bit different. Every control in Windows Forms is a distinct window and therefore has its own hwnd. So, WPF content can be layered

www.it-ebooks.info

anywhere with relation to other Windows Forms controls in the same window, depending on its z-index. (The z-index is determined by the order in which you add controls to the parent's Controls collection, so that controls added later appear on top of those added before.) However, the WPF content still has its own completely distinct region. That means you can't use transparency or any other technique to partially overwrite (or combine your element with) Windows Forms content. Instead, the WPF content exists in its own self-contained region.

Hosting Windows Forms Controls in WPF

To show a Windows Forms control in a WPF window, you use the WindowsFormsHost class in the System.Windows.Forms.Integration namespace. The WindowsFormsHost is a WPF element (it derives from FrameworkElement) that has the ability to hold exactly one Windows Forms control, which is provided in the Child property.

It's easy enough to create and use WindowsFormsHost programmatically. However, in most cases, it's easiest to create it declaratively in your XAML markup. The only disadvantage is that Visual Studio doesn't include much designer support for the WindowsFormsHost control. Although you can drag and drop it onto a window, you need to fill in its content (and map the required namespace) by hand.

The first step is to map the System.Windows.Forms namespace so you can refer to the Windows Forms control you want to use:

```
<Window x:Class="InteroperabilityWPF.HostWinFormControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
    Title="HostWinFormControl" Height="300" Width="300" >
```

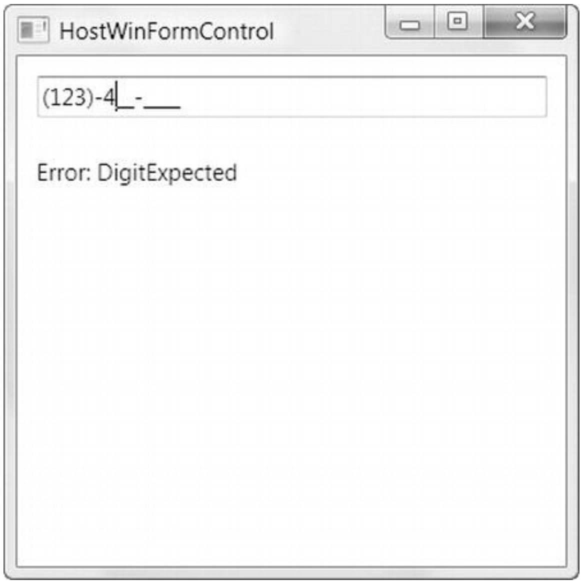
Now you can create the WindowsFormsHost and the control inside just as you would any other WPF element. Here's an example that uses the MaskedTextBox from Windows Forms:

```
<Grid>
  <WindowsFormsHost>
    <wf:MaskedTextBox x:Name="maskedTextBox"></wf:MaskedTextBox>
  </WindowsFormsHost>
</Grid>
```

■ **Note** The WindowsFormsHost can hold any Windows Forms control (that is, any class that derives from System.Windows.Forms.Control). It can't hold Windows Forms components that aren't controls, such as the HelpProvider or the NotifyIcon.

Figure 30-2 shows a MaskedTextBox in a WPF window.

www.it-ebooks.info



(123)-4_-___

Error: DigitExpected

Figure 30-2. *A masked text box for a phone number*

You can set most of the properties of your MaskedTextBox directly in your markup. That's because Windows Forms uses the same TypeConverter infrastructure (discussed in Chapter 2) to change strings into property values of a specific type. This isn't always convenient—for example, the string representation of a type may be awkward to enter by hand—but it usually allows you to configure your Windows Forms controls without resorting to code. For example, here's a MaskedTextBox equipped with a mask that shapes user input into a seven-digit phone number with an optional area code:

```
<wf:MaskedTextBox x:Name="maskedTextBox" Mask="(999)-000-0000"></wf:MaskedTextBox>
```

You can also use ordinary XAML markup extensions to fill in null values, use static properties, create type objects, or use objects that you've defined in the Resources collection of the window. Here's an example that uses the type extension to set the MaskedTextBox.ValidatingType property. This specifies that the MaskedTextBox should change the supplied input (a phone number string) into an Int32 when the Text property is read or the focus changes:

```
<wf:MaskedTextBox x:Name="maskedTextBox" Mask="(999)-000-0000"
  ValidatingType="{x:Type sys:Int32}"></wf:MaskedTextBox>
```

One markup extension that won't work is a data-binding expression because it requires a dependency property. (Windows Forms controls are constructed out of normal .NET properties.) If you want to bind a property of a Windows Forms control to the property of a WPF element, there's an easy workaround—just set the dependency property on the WPF element and adjust the BindingDirection as required. (Chapter 8 has the full details.)

Finally, it's important to note that you can hook events up to your Windows Forms control by using the familiar XAML syntax. Here's an example that attaches an event handler for the MaskInputRejected event, which occurs when a keystroke is discarded because it doesn't suit the mask:

```
<wf:MaskedTextBox x:Name="maskedTextBox" Mask="(999)-000-0000"
  MaskInputRejected="maskedTextBox_MaskInputRejected"></wf:MaskedTextBox>
```

www.it-ebooks.info

Obviously, these aren't routed events, so you can't define them at higher levels in the element hierarchy.

When the event fires, your event handler responds by showing an error message in another element. In this case, it's a WPF label that's located elsewhere on the window:

```
private void maskedTextBox_MaskInputRejected(object sender,
  System.Windows.Forms.MaskInputRejectedEventArgs e)
{
    lblErrorText.Content = "Error: " + e.RejectionHint.ToString();
}
```

■ **Tip** Don't import the Windows Forms namespaces (such as System.Windows.Forms) in a code file that already uses WPF namespaces (such as System.Windows.Controls). The Windows Forms classes and the WPF classes share many names. Basic ingredients (such as Brush, Pen, Font, Color, Size, and Point) and common controls (such as Button, TextBox, and so on) are found in both libraries. To prevent naming clashes, it's best to import just one set of namespaces in your window (WPF namespaces for a WPF window, Windows Forms namespaces for a form) and use fully qualified names or a namespace alias to access the others.

This example illustrates the nicest feature about WPF and Windows Forms interoperability: it doesn't affect your code. Whether you're manipulating a Windows Forms control or a WPF element, you use the familiar class interface for that object. The interoperability layer is simply the magic that lets both ingredients coexist in the window. It doesn't require any extra code.

■ **Note** To have Windows Forms controls use more up-to-date control styles introduced with Windows XP, you must call EnableVisualStyles() when your application starts, as described in the "Visual Styles for Windows Forms Controls" section earlier in this chapter.

Windows Forms content is rendered by Windows Forms, not WPF. Therefore, display-related properties of the WindowsFormsHost container (properties such as Transform, Clip, and Opacity) have no effect on the content inside. This means that even if you set a rotational transform, set a narrow clipping region, and make your content 50 percent transparent, you'll see no change. Similarly, Windows Forms uses a different coordinate system that sizes controls by using physical pixels. As a result, if you increase the system DPI setting of your computer, you'll find that the WPF content resizes cleanly to be more detailed, but the Windows Forms content does not.

Using WPF and Windows Forms User Controls

One of the most significant limitations of the WindowsFormsHost element is that it can hold only a single Windows Forms control. To compensate, you could use a Windows Forms container control. Unfortunately, Windows Forms container controls don't support XAML content models, so you'll need to fill in the contents of the container control programmatically.

A much better approach is to create a Windows Forms user control. This user control can be defined in a separate assembly that you reference, or you can add it directly to your WPF project (using the familiar Add ➤ New Item command). This gives you the best of both worlds—you have full design support to build your user control and an easy way to integrate it into your WPF window.

www.it-ebooks.info

In fact, using a user control gives you an extra layer of abstraction similar to using separate windows. That's because the containing WPF window won't be able to access the individual controls in your user control. Instead, it will interact with the higher-level properties you've added to your user control, which can then modify the controls inside. This makes your code better encapsulated and simpler because it limits the points of interaction between the WPF window and your Windows Forms content. It also makes it easier to migrate to a WPF-only solution in the future, simply by creating a WPF user control that has the same properties and swapping that in place of the WindowsFormsHost. (And once again, you can further improve the design and flexibility of your application by moving the user control into a separate class library assembly.)

■ **Note** Technically, your WPF window can access the controls in a user control by accessing the Controls collection of the user control. However, to use this back door, you need to write error-prone lookup code that searches for specific controls by using a string name. That's always a bad idea.

As long as you're creating a user control, it's a good idea to make it behave as much like WPF content as possible so it's easier to integrate into your WPF window layout. For example, you may want to consider using the FlowLayoutPanel and TableLayoutPanel container controls so that the content inside your user controls flows to fit its dimensions. Simply add the appropriate control and set its Dock property to DockStyle.Fill. Then place the controls you want to use inside. For more information about using the Windows Forms layout controls (which are subtly different from the WPF layout panels), refer to my book *Pro .NET 2.0 Windows Forms and Custom Controls in C#* (Apress, 2005).

ACTIVEX INTEROPERABILITY

WPF has no direct support for ActiveX interoperability. However, Windows Forms has extensive support in the form of *runtime callable wrappers* (RCWs), dynamically generated interop classes that allow a managed Windows Forms application to host an Active component. Although there are .NET-to-COM quirks that can derail some controls, this approach works reasonably well for most scenarios, and it works seamlessly if the person who creates the component also provides a *primary interop assembly*, which is a handcrafted, fine-tuned RCW that's guaranteed to dodge interop issues.

So, how does this help if you need to design a WPF application that uses an ActiveX control? In this case, you need to layer two levels of interoperability. First you place the ActiveX control in a Windows Forms user control or form. You then place that user control in your WPF window or show the form from your WPF application.

Hosting WPF Controls in Windows Forms

The reverse approach—hosting WPF content in a form built with Windows Forms—is just as easy. In this situation, you don't need the WindowsFormsHost class. Instead, you use the System.Windows.Forms.Integration.ElementHost class, which is part of the WindowsFormsIntegration.dll assembly.

The ElementHost has the ability to wrap any WPF element. However, the ElementHost is a genuine Windows Forms control, which means you can place it in a form alongside other Windows Forms content. In some respects, the ElementHost is more straightforward than the WindowsFormsHost, because every control in Windows Forms is displayed as a separate hwnd. Thus, it's not terribly difficult for one of these windows to be rendered with WPF instead of User32/GDI+.

www.it-ebooks.info

Visual Studio provides some design-time support for the ElementHost control, but only if you place your WPF content in a WPF user control. Here's what to do:

1. Right-click the project name in the Solution Explorer, and choose Add ➤ New Item. Pick the User Control (WPF) template, supply a name for your custom component class, and click Add.

■ **Note** This example assumes you're placing the WPF user control directly in your Windows Forms project. If you have a complex user control, you must choose to use a more structured approach and place it in a separate class library assembly.

---

2. Add the WPF controls you need to your new WPF user control. Visual Studio gives you the usual level of design-time support for this step, so you can drag WPF controls from the Toolbox, configure them with the Properties window, and so on.
3. When you're finished, rebuild your project (choose Build ➤ Build Solution). You can't use your WPF user control in a form until you've compiled it.
4. Open to the Windows Forms form where you want to add your WPF user control (or create a new form by right-clicking the project in the Solution Explorer and choosing Add ➤ Windows Form).
5. To place the WPF user control in a form, you need the help of the ElementHost control. The ElementHost control appears on the WPF Interoperability tab of the Toolbox. Drag it onto your form, and size it accordingly.

Tip For better separation, it's a good idea to add the ElementHost to a specific container rather than directly to the form. This makes it easier to separate your WPF content from the rest of the window. Typically, you'll use the Panel, FlowLayoutPanel, or TableLayoutPanel.

---

6. To choose the content for the ElementHost, you use the smart tag. If the smart tag isn't visible, you can show it by selecting the ElementHost and clicking the arrow in the top-right corner. In the smart tag you'll find a drop-down list named Select Hosted Content. Using this list, you can pick the WPF user control you want to use, as shown in Figure 30-3.

www.it-ebooks.info

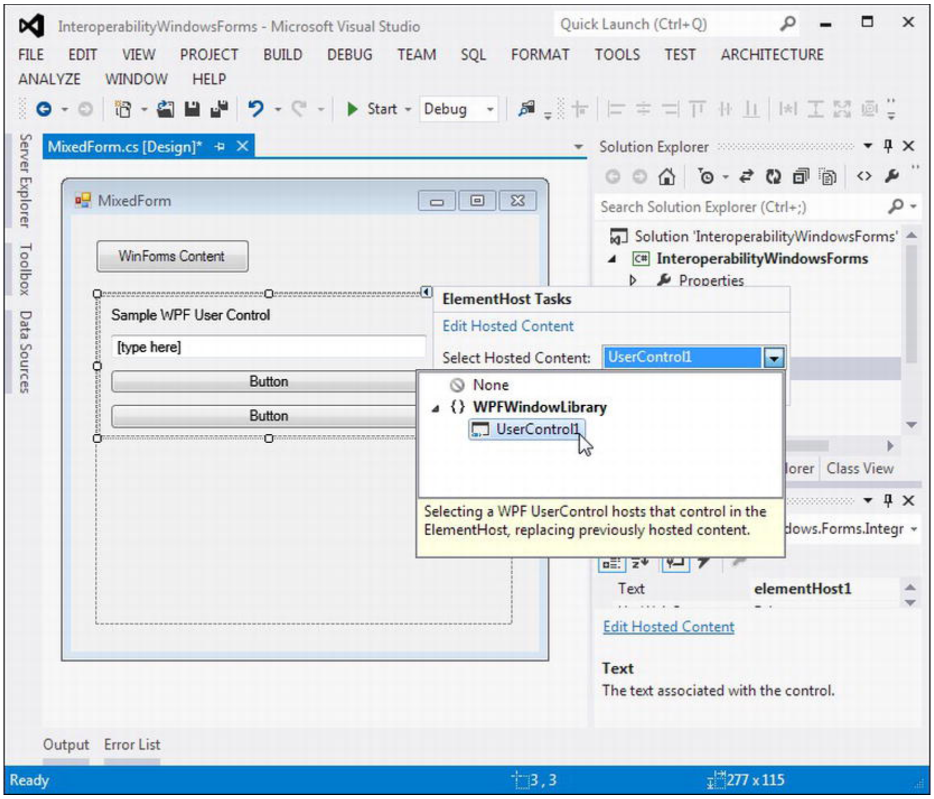

InteroperabilityWindowsForms - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL FORMAT TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Start Debug
MixedForm.cs [Design]*
Server Explorer Toolbox Data Sources
MixedForm
WinForms Content
Sample WPF User Control
[type here]
Button
Button
ElementHost Tasks
Edit Hosted Content
Select Hosted Content: UserControl1
None
WPFWindowLibrary
UserControl1
Selecting a WPF UserControl hosts that control in the ElementHost, replacing previously hosted content.
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'InteroperabilityWindowsForms'
InteroperabilityWindowsForms
Properties
Class View
Text elementHost1
Edit Hosted Content
Text
The text associated with the control.
Output Error List
Ready 3 , 3 277 x 115

Figure 30-3. *Selecting WPF content for an ElementHost*

7. Although the WPF user control will appear in your form, you can't edit its content there. To jump to the corresponding XAML file in a hurry, click the Edit Hosted Content link in the ElementHost smart tag.

Technically, the ElementHost can hold any type of WPF element. However, the ElementHost smart tag expects you to choose a user control that's in your project (or a referenced assembly). If you want to use a different type of control, you'll need to write code that adds it to the ElementHost programmatically.

Access Keys, Mnemonics, and Focus

The WPF and Windows Forms interoperability works because the two types of content can be rigorously separated. Each region handles its own rendering and refreshing and interacts with the mouse independently. However, this segregation isn't always appropriate. For example, it runs into potential problems with keyboard handling, which sometimes needs to be global across an entire form. Here are some examples:

- When you tab from the last control in one region, you expect focus to move to the first control in the next region.
- When you use a shortcut key to trigger a control (such as a button), you expect that button to respond no matter what region of the window it's located in.

www.it-ebooks.info

- When you use a label mnemonic, you expect the focus to move to the linked control.
- Similarly, if you suppress a keystroke by using a preview event, you don't expect the corresponding key event to occur in either region, no matter what control currently has focus.

The good news is that all these expected behaviors work without any customization needed. For example, consider the WPF window shown in Figure 30-4. It includes two WPF buttons (top and bottom) and a Windows Forms button (in the middle).

Here's the markup:

```
<Grid>
```

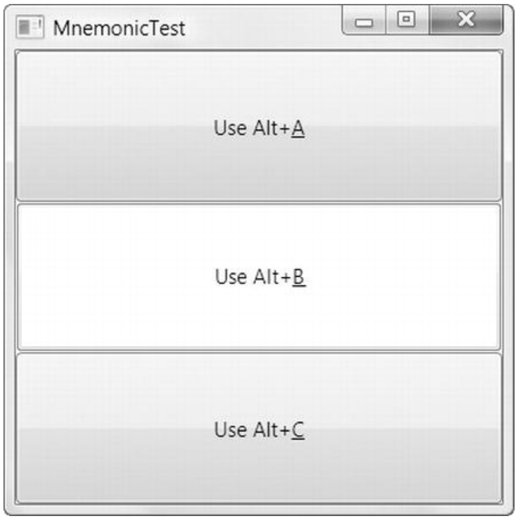


Figure 30-4. *Three buttons with shortcut keys*

```
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Button Click="cmdClicked">Use Alt+_A</Button>
  <WindowsFormsHost Grid.Row="1">
    <wf:Button Text="Use Alt+&amp;B" Click="cmdClicked"></wf:Button>
  </WindowsFormsHost>
  <Button Grid.Row="2" Click="cmdClicked">Use Alt+_C</Button>
</Grid>
```

■ **Note** The syntax for identifying accelerator keys is slightly different in WPF (which uses an underscore) than in Windows Forms. Windows Forms uses the & character, which must be escaped as &amp; in XML because it's a special character.

www.it-ebooks.info

When this window first appears, the text in all buttons is normal. When the user presses and holds the Alt key, all three shortcuts are underlined. The user can then trigger any one of the three buttons by pressing the A, B, or C key (while holding down Alt).

The same magic works with mnemonics, which allows labels to forward the focus to a nearby control (typically a text box). You can also tab through the three buttons in this window as though they were all WPF-defined controls, moving from top to bottom. Finally, the same example continues to work if you host a combination of Windows Forms and WPF content in a Windows Forms form.

Keyboard support isn't always this pretty, and there are a few focus-related quirks that you may run into. Here's a list of issues to watch out for:

- Although WPF supports a keystroke forwarding system to make sure every element and control gets a chance to handle keyboard input, the keyboard-handling models of WPF and Windows Forms still differ. For that reason, you won't receive keyboard events from the WindowsFormsHost when the focus is in the Windows Forms content inside. Similarly, if the user moves from one control to another inside a WindowsFormsHost, you won't receive the GotFocus and LostFocus events from the WindowsFormsHost.

■ **Note** Incidentally, the same is true for WPF mouse events. For example, the MouseMove event won't fire for the WindowsFormsHost while you move the mouse inside its bounds.

- Windows Forms validation won't fire when you move the focus from a control inside the WindowsFormsHost to an element outside the WindowsFormsHost. Instead, it will fire only when you move from one control to another inside the WindowsFormsHost. (When you remember that the WPF content and the Windows Forms content are essentially separated windows, this makes perfect sense, because it's the same behavior you experience if you switch between different applications.)
- If the window is minimized while the focus is somewhere inside a WindowsFormsHost, the focus may not be restored when the window is restored.

Property Mapping

One of the most awkward details in interoperability between WPF and Windows Forms is the way they use similar but different properties. For example, WPF controls have a Background property that allows you to supply a brush that paints the background. Windows Forms controls use a simpler BackColor property that fills the background with a color based on an ARGB value. Obviously, there's a disconnect between these two properties, even though they're often used to set the same aspect of a control's appearance.

Most of the time, this isn't a problem. As a developer, you'll simply be forced to switch between both APIs, depending on the object you're working with. However, WPF adds a little bit of extra support through a feature called *property translators*.

Property translators won't allow you to write WPF-style markup and have it work with Windows Forms controls. In fact, property translators are quite modest. They simply convert a few basic properties of the WindowsFormsHost (or ElementHost) from one system to another so that they can be applied on the child control.

For example, if you set the WindowsFormsHost.IsEnabled property, the Enabled property of the control inside is modified accordingly. This isn't a necessary feature (you could do much the same thing by

www.it-ebooks.info

modifying the Enabled property of the child directly, instead of the IsEnabled property of the container), but it can often make your code a bit clearer.

To make this work, the WindowsFormsHost and ElementHost classes both have a PropertyMap collection, which is responsible for associating a property name with a delegate that identifies a method that performs the conversion. By using a method, the property map system is able to handle sticky conversions such as BackColor to Background, and vice versa. By default, each is filled with a default set of associations. (You're free to create your own or replace the existing ones, but this degree of low-level fiddling seldom makes sense.)

Table 30-1 lists the standard property map conversions that are provided by the WindowsFormHost and ElementHost classes.

Table 30-1. *Property Maps*

WPF Property	Windows Forms Property	Comments
Foreground	ForeColor	Converts any ColorBrush into the corresponding Color object. In the case of a GradientBrush, the color of the GradientStop with the lowest offset value is used instead. For any other type of brush, the ForeColor is not changed, and the default is used.
Background	BackColor or BackgroundImage	Converts any SolidColorBrush to the corresponding Color object. Transparency is not supported. If a more exotic brush is used, the WindowsFormsHost creates a bitmap and assigns it to the BackgroundImage property instead.
Cursor	Cursor	
FlowDirection	RightToLeft	
FontFamily, FontSize, FontStretch, FontStyle, FontWeight	Font	
IsEnabled	Enabled	
Padding	Padding	
Visibility	Visible	Converts a value from the Visibility enumeration into a Boolean value. If Visibility is Hidden, the Visible property is set to true so that the content size can be used for layout calculations but the WindowsFormsHost does not draw the content. If Visibility is Collapsed, the Visible property is not changed (so it remains with its currently set or default value), and the WindowsFormsHost does not draw the content.

www.it-ebooks.info

■ **Note** Property maps work dynamically. For example, if the WindowsFormsHost.FontFamily property is changed, a new Font object is constructed and applied to the Font property of the child control.

WIN32 INTEROPERABILITY

WPF certainly doesn't limit its interoperability to Windows Forms applications—if you want to work with the Win32 API or place WPF content in a C++ MFC application, you can do that too.

You can host Win32 in WPF by using the System.Windows.Interop.HwndHost class, which works analogously to the WindowsFormsHost class. The same limitations that apply to WindowsFormsHost apply to HwndSource (for example, the airspace rule, focus quirks, and so on). In fact, WindowsFormsHost derives from HwndHost.

The HwndHost is your gateway to the traditional world of C++ and MFC applications. However, it also allows you to integrate managed DirectX content. Currently, WPF does not include any DirectX interoperability features, and you can't use the DirectX libraries to render content in a WPF window. However, you can use DirectX to build a separate window and then host that inside a WPF window by using the HwndHost. Although DirectX is far beyond the scope of this book (and an order of magnitude more complex than WPF programming), you can learn more at `http://msdn.microsoft.com/directx`.

The complement of HwndHost is the HwndSource class. While HwndHost allows you to place any hwnd in a WPF window, HwndSource wraps any WPF visual or element in an hwnd so it can be inserted in a Win32-based application, such as an MFC application. The only limitation is that your application needs a way to access the WPF libraries, which are managed .NET code. This isn't a trivial task. If you're using a C++ application, the simplest approach is to use the Managed Extensions for C++. You can then create your WPF content, create an HwndSource to wrap it, set the HwndHost.RootVisual property to the top-level element, and then place the HwndSource into your window.

The Last Word

In this chapter you considered the interoperability support that allows WPF applications to show Windows Forms content (and vice versa). Then you examined the WindowsFormsHost element, which lets you embed a Windows Forms control in a WPF window, and the ElementHost, which lets you embed a WPF element in a form. Both of these classes provide a simple, effective way to manage the transition from Windows Forms to WPF.

www.it-ebooks.info