CHAPTER 25

# Menus, Toolbars, and Ribbons

A few rich controls can appear in virtually any type of application, from document editors to system utilities. Those are the controls that you'll meet in this chapter. They include the following:

- *Menus.* They're one of the oldest user interface controls, and they've changed surprisingly little in the past two decades. WPF includes solid, straightforward support for main menus and pop-up context menus.
- *Toolbars and status bars.* They decorate the top and bottom of countless applications—sometimes when they aren't even needed. WPF supports both controls with its customary flexibility, allowing you to insert virtually any control inside. However, the WPF toolbars and status bars don't have many frills. They support overflow menus, but they don't provide floating and docking capability.
- *Ribbons.* With only a little more effort, you can add an Office-style ribbon to the top of your application window. It requires a separate (free) download, but you'll get some valuable built-in features, such as configurable resizing. You'll also get an Office-style menu feature to match.

---

**What's New** WPF 4.5 adds native Office ribbon functionality. However, it only works for Office applications. In other words, you can use WPF 4.5 to build an Office add-in that extends the ribbon in Word, Excel, PowerPoint, Visio, Outlook, InfoPath, or Project. (For more information, see `http://tinyurl.com/945vpsj`.) However, if you want to add an Office-style ribbon to a custom desktop application, you need to download an extra component, as described in this chapter.

---

## Menus

WPF provides two menu controls: Menu (for main menus) and ContextMenu (for pop-up menus that are attached to other elements). Like all the WPF classes, WPF performs the rendering for the Menu and ContextMenu controls. That means these controls aren't simple wrappers around Windows libraries. This gives them more flexibility, including the ability to be used in browser-hosted applications.

■ **Note** If you use the Menu class in a browser-hosted application, it appears at the top of the page. The browser window wraps your page, and it may or may not include a menu of its own, which will be completely separate.

## The Menu Class

WPF doesn't make any assumption about where a stand-alone menu should be placed. Ordinarily, you'll dock it at the top of your window using a DockPanel or the top row of a Grid, and you'll stretch it across the entire width of your window. However, you can place a menu anywhere, even alongside other controls (as shown in Figure 25-1). Furthermore, you can add as many menus in a window as you want. Although it might not make much sense, you have the ability to stack menu bars or scatter them throughout your user interface.

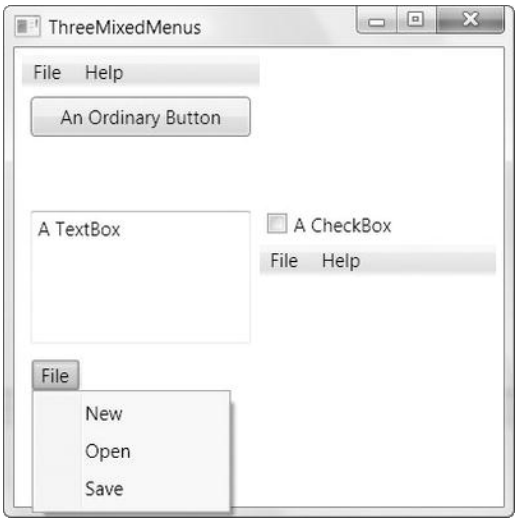


***Figure 25-1.*** *Mixed menus*

This freedom provides some interesting possibilities. For example, if you create a menu with one top-level heading and style it to look like button, you'll end up with a one-click pop-up menu (like the menu that's activated in Figure 25-1). This sort of user interface trickery might help you get the exact effect you want in a highly customized interface. Or, it might just be a more powerful way to confuse users.

The Menu class adds a single new property: IsMainMenu. When true (which is the default value), pressing the Alt key or F10 gives the menu focus, just as in any other Windows application. Along with this small detail, the Menu container has a few of the familiar ItemsControl properties for you to play with. That means you can create data-bound menus using the ItemsSource DisplayMemberPath, ItemTemplate, and ItemTemplateSelector properties. You can also apply grouping, change the layout of menu items inside the menu, and apply styles to your menu items.

For example, Figure 25-2 shows a scrollable sidebar menu. You can create it by supplying a StackPanel for the ItemsPanel property, changing its background, and wrapping the entire Menu in a ScrollViewer. Obviously, you can make more radical changes to the visual appearance of menus and submenus using

triggers and control templates. The bulk of the styling logic is in the default control template for the MenuItem.

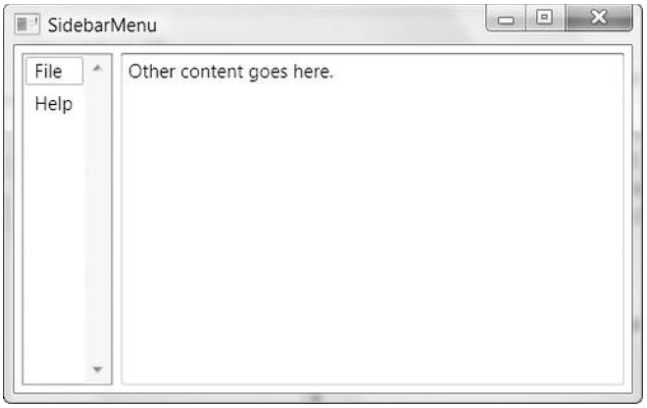


***Figure 25-2.*** *A menu in a StackPanel*

## Menu Items

Menus are composed of MenuItem objects and Separator objects. The MenuItem class derives from HeaderedItemsControl, because each menu item has a header (which contains the text for that item) and can hold a collection of MenuItem objects (which represents a submenu). The Separator simply displays a horizontal line separating menu items.

Here's a straightforward combination of MenuItem objects that creates the rudimentary menu structure shown in Figure 25-3:

```
<Menu>
  <MenuItem Header="File">
    <MenuItem Header="New"></MenuItem>
    <MenuItem Header="Open"></MenuItem>
    <MenuItem Header="Save"></MenuItem>
    <Separator></Separator>
    <MenuItem Header="Exit"></MenuItem>
  </MenuItem>
  <MenuItem Header="Edit">
    <MenuItem Header="Undo"></MenuItem>
    <MenuItem Header="Redo"></MenuItem>
    <Separator></Separator>
    <MenuItem Header="Cut"></MenuItem>
    <MenuItem Header="Copy"></MenuItem>
    <MenuItem Header="Paste"></MenuItem>
  </MenuItem>
</Menu>
```

As with buttons, you can use the underscore to indicate an Alt+ shortcut key combination. Whereas this is often considered an optional feature in buttons, most menu users expect to have keyboard shortcuts.

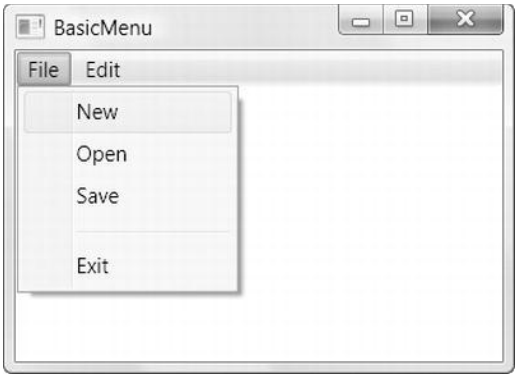


***Figure 25-3.*** *A basic menu*

WPF allows you to break most of the commonsense rules of structuring a menu. For example, you can have non-MenuItem objects inside a Menu or MenuItem. This allows you to create menus that hold ordinary WPF elements, ranging from the ordinary CheckBox to a DocumentViewer. For a variety of reasons, placing non-MenuItem objects in a menu is almost always a bad way to go. If you place non-MenuItem objects in a menu, they'll exhibit a few oddities that you'll need to track down and correct. For example, a TextBox in a MenuItem will lose focus as soon as you move the mouse out of the bounds of the MenuItem. If you really want a user interface that includes some sort of drop-down menu with controls, consider using another element (such as the Expander) and styling it to suit your needs. Use menus only when you really want the behavior of a menu—in other words, a group of clickable commands.

---

■ **Note** Set the MenuItem.StaysOpenOnClick property to true if you want submenus to remain visible when opened until the user clicks somewhere else.

---

MenuItem objects can also be used *outside* the standard Menu, ContextMenu, and MenuItem containers. These items behave just like ordinary menu items—they glow blue when you hover over them, and they can be clicked to trigger actions. However, any submenus they include won't be accessible. Again, this is an aspect of Menu flexibility you probably won't want to use.

To react when a MenuItem is clicked, you may choose to handle the MenuItem.Click event. You can handle it for individual items, or you can attach an event handler to the root Menu tag. Your other alternative is to use the Command, CommandParameter, and CommandTarget properties to connect a MenuItem to a Command object, as you learned to do with buttons in Chapter 9. This is particularly useful if your user interface includes multiple menus (for example, a main menu and a context menu) that use the same commands or includes a menu and a toolbar that do.

Along with text content (which is supplied through the Header property), MenuItem objects can actually show several more details:

- A thumbnail icon in the margin area just to the left of the menu command.
- A check mark in the margin area. If you set the check mark and an icon, only the check mark appears.
- Shortcut text to the right of the menu text. For example, you might see Ctrl+O to indicate the shortcut key for the Open command.

Setting all these ingredients is easy. To show a thumbnail icon, you set the MenuItem.Icon property. Interestingly, the Icon property accepts any object, which gives you the flexibility to construct a miniature vector drawing. This way, you can take full advantage of WPF's resolution-independent scaling to show more detail at higher system DPI settings. If you want to use an ordinary icon, simply use an Image element with a bitmap source.

To show a check mark next to a menu item, you simply need to set the MenuItem.IsChecked property to true. Additionally, if IsCheckable is true, clicking the menu item will toggle back and forth between its checked and unchecked state. However, there's no way to associate a group of checked menu items. If that's the effect you want, you need to write the code to clear the other check boxes when an item is checked.

You can set the shortcut text for a menu item using the MenuItem.InputGestureText property. However, simply displaying this text doesn't make it active. It's up to you to watch for the key presses you want. This is almost always too much work, so menu items are commonly used with commands, which gives you the shortcut key behavior and the InputGestureText in one step.

For example, the following MenuItem is linked to the ApplicationsCommands.Open command:

```
<MenuItem Command="ApplicationCommands.Open"></MenuItem>
```

This command already has the Ctrl+O keystroke defined in the RoutedUICommand.InputGestures command collection. As a result, Ctrl+O appears for the shortcut text, and the Ctrl+O keystroke triggers the command (assuming you've wired up the appropriate event handler). If a keystroke wasn't defined, you could add it to the InputGestures collection yourself.

---

■ **Tip** Several useful properties indicate the current state of the MenuItem, including IsChecked, IsHighlighted, IsPressed, and IsSubmenuOpen. You can use these to write triggers that apply different styling in response to certain actions.

---

## The ContextMenu Class

Like the Menu, the ContextMenu class holds a collection of MenuItem objects. The difference is that a ContextMenu can't be placed in a window. Instead, it can be used only to set the ContextMenu property of another element:

```
<TextBox>
  <TextBox.ContextMenu>
    <MenuItem ... >
     ...
    </MenuItem>
  </TextBox.ContextMenu>
</TextBox>
```

The ContextMenu property is defined in the FrameworkElement class, so it's supported by virtually all WPF elements. If you set the ContextMenu property of an element that ordinarily has its own context menu, your menu replaces the standard menu. If you simply want to remove an existing context menu, just set it to a null reference.

When you attach a ContextMenu object to an element, it appears automatically when the user right-clicks that control (or presses Shift+F10 while it has focus). The context menu won't appear if the element has IsEnabled set to false, unless you explicitly allow this with the ContextMenuService.ShowOnDisabled attached property:

```
<TextBox ContextMenuService.ShowOnDisabled="True">
  <TextBox.ContextMenu>
     ...
  </TextBox.ContextMenu>
</TextBox>
```

## Menu Separators

The Separator is a standard element for dividing menus into groups of related commands. However, the content of the separator is completely fluid, thanks to control templates. By taking a separator and supplying a new template, you can add other, nonclickable elements to your menus, such as subheadings.

You might expect that you could add a subheading simply by adding a non-MenuItem object to a menu, such as a TextBlock with some text. However, if you take this step, the newly added element keeps the menu selection behavior; this means you can step through it with the keyboard, and when you hover over it with the mouse, the edges glow blue. The Separator doesn't exhibit this behavior—it's a fixed piece of content that doesn't react to keyboard or mouse actions.

Here's an example of a Separator that defines a text title:

```
<Separator>
  <Separator.Template>
    <ControlTemplate>
      <Border CornerRadius="2" Padding="5" Background="PaleGoldenrod"
       BorderBrush="Black" BorderThickness="1">
        <TextBlock FontWeight="Bold">
         Editing Commands
        </TextBlock>
      </Border>
      </ControlTemplate>
    </Separator.Template>
</Separator>
```

Figure 25-4 shows the title this creates.


***Figure 25-4.*** *A menu that includes a fixed subheading*

Unfortunately, the Separator isn't a content control, so it's not possible to separate the content you want to show (for example, the string of text) from the formatting you want to use. That means you'll be forced to define the same template each time you use the separator if you want to vary its text. To make this process a bit simpler, you can create a separator style that bundles together all the properties you want to set on the TextBlock inside the Separator, except for the text.

# Toolbars and Status Bars

Toolbars and status bars are two well-worn staples of the Windows world. Both are specialized containers that hold a collection of items. Traditionally, a toolbar holds buttons, and a status bar consists primarily of text and other noninteractive indicators (like a progress bar). However, both toolbars and status bars are used with a variety of different controls.

In Windows Forms, toolbars and status bars have their own content model. Although it's still possible to place arbitrary controls inside a toolbar and status bar using a wrapper, the process isn't seamless. The WPF toolbar and status bar don't have this limitation. They support the WPF content model, allowing you to add any element to a toolbar or status bar and giving you unparalleled flexibility. In fact, there are no toolbar-specific or status bar–specific elements. Everything you need is already available in the basic collection of WPF elements.

## The ToolBar

A typical WPF ToolBar is filled with Button, ComboBox, CheckBox, RadioButton, and Separator objects. Because these elements are all content controls (except for the Separator), you can place text and image content inside. Although you can use other elements, such as Label and Image to put noninteractive elements into the ToolBar, the effect is often confusing.

At this point, you might be wondering how you can place these common controls in a toolbar without creating an odd visual effect. After all, the content that appears in standard Windows toolbars looks quite a bit different from similar content that appears in a window. For example, the buttons in a toolbar are displayed with a flat, streamlined appearance that removes the border and the shaded background. The toolbar surface shows through underneath, and the button glows blue when you hover over it with the mouse.

In the WPF way of thinking, the button in a toolbar is the same as a button in a window—both are clickable regions you can use to perform an action. The only difference is the visual appearance. Thus, the perfect solution is to use the existing Button class but adjust various properties or change the control template. This is exactly what the ToolBar class does—it overrides the default style of some types of children, including the buttons. You can still have the last word by manually setting the Button.Style property if you want to create your own customized toolbar button, but usually you'll get all the control you need by setting the button content.

Not only does the ToolBar change the appearance of many of the controls its holds, but it also changes the behavior of the ToggleButton and the CheckBox and RadioButton that derive from it. A ToggleButton or CheckBox in a ToolBar is rendered like an ordinary button, but when you click it, the button remains highlighted (until you click it again). The RadioButton has a similar appearance, but you must click another RadioButton in a group to clear the highlighting. (To prevent confusion, it's always best to separate a group of RadioButton objects in a toolbar using the Separator.)

To demonstrate what this looks like, consider the simple markup shown here:

```
<ToolBar>
  <Button Content="{StaticResource DownloadFile}"></Button>
  <CheckBox FontWeight="Bold">Bold</CheckBox>
```

```
  <CheckBox FontStyle="Italic">Italic</CheckBox>
  <CheckBox>
    <TextBlock TextDecorations="Underline">Underline</TextBlock>
  </CheckBox>
  <Separator></Separator>
  <ComboBox SelectedIndex="0">
    <ComboBoxItem>100%</ComboBoxItem>
    <ComboBoxItem>50%</ComboBoxItem>
    <ComboBoxItem>25%</ComboBoxItem>
  </ComboBox>
  <Separator></Separator>
</ToolBar>
```

Figure 25-5 shows this toolbar in action, with two CheckBox controls in the checked state and the drop-down list on display.


***Figure 25-5.*** *Different controls in a toolbar*

Although the example in Figure 25-5 is limited to buttons that contain text, ToolBar buttons usually hold image content. (You can also combine both by wrapping an Image element and a TextBlock or Label in a horizontal StackPanel.) If you're using image content, you need to decide whether you want to use bitmap images (which may show scaling artifacts at different resolutions), icons (which improve this situation somewhat because you can supply several differently sized images in one file), or vector images (which require the most markup but provide flawless resizing).

The ToolBar control has a few oddities. First, unlike other controls that derive from ItemsControl, it doesn't supply a dedicated wrapper class. (In other words, there is a ToolBarItem class.) The ToolBar simply doesn't require this wrapper to manage items, track selection, and so on, as other list controls. Another quirk in the ToolBar is that it derives from HeaderedItemsControl even though the Header property has no effect. It's up to you to use this property in some interesting way. For example, if you have an interface that uses several ToolBar objects, you could allow users to choose which ones to display from a context menu. In that menu, you could use the toolbar name that's set in the Header property.

The ToolBar has one more interesting property: Orientation. You can create a top-to-bottom toolbar that's docked to one of the sides of your window by setting the ToolBar.Orientation property to Vertical. However, each element in the toolbar will still be oriented horizontally (for example, text won't be turned on its side), unless you use a LayoutTransform to rotate it.

## The Overflow Menu

If a toolbar has more content than it can fit in a window, it removes items until the content fits. These extra items are placed into an overflow menu, which you can see by clicking the drop-down arrow at the end of

the toolbar. Figure 25-6 shows the same toolbar shown in Figure 25-5 but in a smaller window that necessitates an overflow menu.

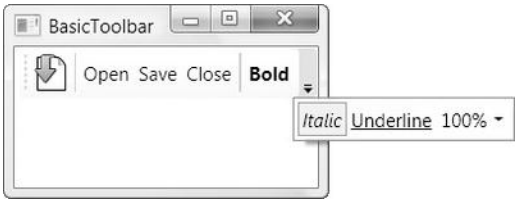


***Figure 25-6.*** *The automatic overflow menu*

The ToolBar control adds items to the overflow menu automatically, starting with the last item. However, you can configure the way this behavior works to a limited degree by applying the attached ToolBar.OverflowMode property to the items in the toolbar. Use OverflowMode.Never to ensure that an important item is never placed in the overflow menu, OverflowMode.AsNeeded (the default) to allow it to be placed in the overflow menu when space is scarce, or OverflowMode.Always to force an item to remain permanently in the overflow menu.

---

■ **Note** Always items, the items that don't fit will be clipped off at the bounds of the container and will be inaccessible to the user.

---

If your toolbar contains more than one OverflowMode.AsNeeded item, the ToolBar removes items that are at the end of the toolbar first. Unfortunately, there's no way to assign relative priorities to toolbar items. For example, there's no way to create an item that's allowed in the overflow menu but won't be placed there until every other relocatable item has already been moved. There's also no way to create buttons that adapt their sizes based on the available space, as you can with the ribbon discussed later in this chapter.

## The ToolBarTray

Although you're free to add multiple ToolBar controls to your window and manage them using a layout container, WPF has a class that's designed to take care of some of the work: the ToolBarTray. Essentially, the ToolBarTray holds a collection of ToolBar objects (which are exposed through a property named ToolBars).

The ToolBarTray makes it easier for toolbars to share the same row, or *band*. You can configure the ToolBarTray so that toolbars share a band, while others are placed on other bands. The ToolBarTray provides the shaded background behind the entire ToolBar area. But most important, the ToolBarTray adds support for toolbar drag-and-drop functionality. Unless you set the ToolBarTray.IsLocked property to true, the user can rearrange your toolbars in a ToolBar tray by clicking the grip at the left side. Toolbars can be repositioned in the same band or moved to a different band. However, the user is not able to drag a toolbar from one ToolBarTray to another. If you want to lock down individual toolbars, simply set the ToolBarTray.IsLocked attached property on the appropriate ToolBar objects.

---

■ **Note** When moving toolbars, it's possible that some content may be obscured. For example, the user may move a toolbar to a position that leaves very little room for another adjacent toolbar. In this situation, the missing items are added to the overflow menu.

---

You can place as many ToolBar objects as you want in a ToolBarTray. By default, all your toolbars will be placed in left-to-right order on the topmost band. Initially, each toolbar is given its full desired width. (If a subsequent toolbar doesn't fit, some or all of its buttons are moved to the overflow menu.) To get more control, you can specify which band a toolbar should occupy by setting the Band property using a numeric index (where 0 is the topmost band). You can also set the placement inside the band explicitly by using the BandIndex property. A BandIndex of 0 puts the toolbar at the beginning of the band.

Here's some sample markup that creates several toolbars in a ToolBarTray. Figure 25-7 shows the result.

```
<ToolBarTray>
  <ToolBar>
    <Button>One</Button>
    <Button>Two</Button>
    <Button>Three</Button>
  </ToolBar>
  <ToolBar>
    <Button>A</Button>
    <Button>B</Button>
    <Button>C</Button>
  </ToolBar>
  <ToolBar Band="1">
    <Button>Red</Button>
    <Button>Blue</Button>
    <Button>Green</Button>
    <Button>Black</Button>
  </ToolBar>
</ToolBarTray>
```


***Figure 25-7.*** *Grouping toolbars in the ToolBarTray*

## The StatusBar

Compared to the ToolBar, the StatusBar is a much less glamorous control class. Like the ToolBar, it holds any content (which it wraps implicitly in StatusBarItem objects), and it overrides the default styles of some elements to provide more suitable rendering. However, the StatusBar control doesn't have the support for draggable rearranging or an overflow menu. It's primarily used to display text and image indicators (and the occasional progress bar).

The StatusBar doesn't work very well if you want to use one of the ButtonBase-derived elements or the ComboBox. It doesn't override the styles of any of these controls, so they look out of place in the status bar. If you need to create a status bar that includes these controls, you might consider docking an ordinary ToolBar control to the bottom of your window. It's probably as a result of this general lack of features that the StatusBar is found in the System.Windows.Controls.Primitives namespace rather than in the more mainstream System.Windows.Controls namespace where the ToolBar control exists.

There's one tip worth noting if you're using a status bar. Ordinarily, the StatusBar control lays its children out from left to right using a horizontal StackPanel. However, applications often use proportionately sized status bar items or keep items locked to the right side of the status bar. You can implement this design by specifying that the status bar should use a different panel using the ItemsPanelTemplate property, which you first considered in Chapter 20.

One way to get proportionally or right-aligned items is to use a Grid for your layout container. The only trick is that you must wrap the child element in a StatusBarItem object in order to set the Grid.Column property appropriately. Here's an example that uses a Grid to place one TextBlock on the left side of a StatusBar and another on the right side:

```
<StatusBar Grid.Row="1">
  <StatusBar.ItemsPanel>
    <ItemsPanelTemplate>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="*"></ColumnDefinition>
          <ColumnDefinition Width="Auto"></ColumnDefinition>
        </Grid.ColumnDefinitions>
      </Grid>
    </ItemsPanelTemplate>
  </StatusBar.ItemsPanel>
  <TextBlock>Left Side</TextBlock>
  <StatusBarItem Grid.Column="1">
    <TextBlock>Right Side</TextBlock>
  </StatusBarItem>
</StatusBar>
```

This highlights one of the key advantages of WPF—other controls can benefit from the core layout model without needing to re-create it. By contrast, Windows Forms included several controls that wrapped some sort of proportionally sized items, including the StatusBar and the DataGridView. Despite the conceptual scenario, these controls were forced to include their own layout model and add their own layout-specific properties to manage child items. In WPF, this isn't the case—every control that derives from ItemsControl can use any panel to arrange its child items.

# Ribbons

At this point, you might be feeling that the WPF toolbars are just a bit underwhelming. Other than two built-in features—a basic overflow menu and the ability to be rearranged by the user—they don't provide any modern frills. Even the Windows Forms toolkit has a feature that allows users to drag and dock toolbars to different places in a window.

The reason that toolbars haven't evolved since the first version of WPF is simple: they're a dying trend. Although toolbars are still relatively popular at the moment, the shift is to smarter tab-based controls, such as the ribbon that debuted in Office 2007 and now graces Windows Explorer (in Windows 8) and Office 2013.

With the ribbon, Microsoft found itself faced with a familiar dilemma. To improve the productivity and consistency of all Windows applications, Microsoft wanted to encourage every application to adopt the ribbon. But because Microsoft also wanted to keep its competitive edge, it wasn't in a rush to release the APIs that would make that possible. After all, Microsoft spent thousands of hours of research and development in perfecting its version of the ribbon, so it's no surprise that the company took a few years to enjoy the result.

Fortunately, the wait has ended, and Microsoft has made a version of the ribbon available to WPF developers. The good news is that it's completely free and respectably full-featured, including rich tooltips, drop-down buttons, dialog launchers, a quick access toolbar, and configurable resizing.

However, the ribbon control isn't included with the .NET Framework. Instead, you need to download it from Microsoft's Download Center. Just search for "WPF ribbon" at `http://www.microsoft.com/download`. At the time of this writing, you can download the latest version at `http://tinyurl.com/8aphzsf`. Click "Microsoft Ribbon for WPF.msi" to install the compiled class library that houses the ribbon control, which is named RibbonControlsLibrary.dll. You can also click "Microsoft Ribbon for WPF Source and Samples.msi" to install sample projects that use the ribbon, including one that mimics the Home tab in the user interface for Word.

---

■ **Tip** Before you begin using the ribbon, it's worth reviewing the design guidelines and best practices for the ribbon, which you can read at `http://tinyurl.com/4dsbef`.

---

## Adding the Ribbon

To start using the ribbon, start by creating a new WPF project and adding a reference to the RibbonControlsLibrary.dll assembly. You'll find it in a folder like Program Files (x86)\Microsoft Ribbon for WPF\V4.0.

As with any control that's not a part of the core WPF libraries, you need to map the control assembly to an XML prefix before you can use it:

```
<Window x:Class="RibbonTest.MainWindow" ... xmlns:r=
 "clr-namespace:Microsoft.Windows.Controls.Ribbon;assembly=RibbonControlsLibrary">
```

You can then add an instance of the Ribbon control anywhere in your window:

```
<r:Ribbon>
</r:Ribbon>
```

By far the best position for the ribbon is at the top of the window, using a Grid or Dock panel. But before you go any further, there's one change worth making. The RibbonControlsLibrary.dll assembly includes a RibbonWindow class—a class that derives from Window but provides additional ribbon integration features. Most significantly, the RibbonWindow class provides a spot for the quick access toolbar (which is a customizable group of commonly used buttons that's displayed above the ribbon), and a spot to put the header for contextual ribbon tabs (these are tabs that only appear for certain tasks). Figure 25-8 compares the difference, with the normal window on the left and the RibbonWindow on the right.

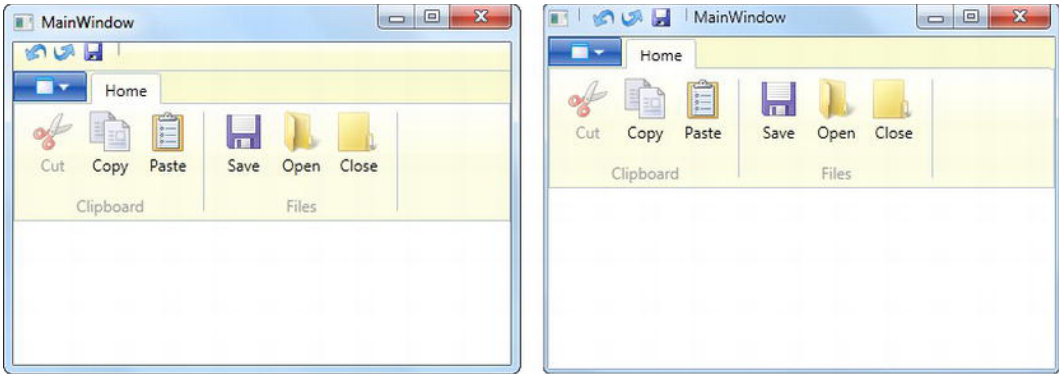


***Figure 25-8.*** *Putting the ribbon in a RibbonWindow*

You'll learn about the quick access toolbar later in this chapter. For now, you can use the outline of a basic ribbon-enabled window shown in the following code. This custom window derives from RibbonWindow and places the ribbon at the top, while reserving the second row of a Grid for the actual window content.

```
<r:RibbonWindow x:Class="RibbonTest.MainWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="MainWindow" Height="350" Width="525"
 xmlns:r=
 "clr-namespace:Microsoft.Windows.Controls.Ribbon;assembly=RibbonControlsLibrary">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>

    <r:Ribbon>
    </r:Ribbon>
  </Grid>
</r:RibbonWindow>
```

This assumes that your window is named MainWindow and your project is named RibbonTest—you'll need to adjust these details in the class declaration to match your names.

When using the RibbonWindow, make sure your code-behind window class doesn't explicitly derive from Window. If it does, change the inherited class to RibbonWindow. Or, remove that part of the class declaration altogether, as shown here:

```
public partial class MainWindow
{ ... }
```

This works because the automatically generated portion of the MainWindow class already has the right RibbonWindow derivation, because it's specified in the XAML.

The Ribbon control actually consists of three pieces: the quick access toolbar (which sits at the top), an application menu (which is exposed through the button on the far left, before any tabs), and the multitabbed ribbon itself. In the following sections, you'll explore all three parts.

■ **Tip** If you don't like the ribbon's blue color, you can set the Background property with the color of your choice. There's no need to use a fancy gradient brush, either, because the ribbon takes the solid color you supply and automatically adds a subtle gradient effect over the surface of the control.

## The Application Menu

The easiest way to get started with the ribbon is to fill the application menu.

The application menu is based on two straightforward classes: RibbonApplicationMenu (which derives from MenuBase) and RibbonApplicationMenuItem (which derives from MenuItem). This establishes a pattern you'll see throughout this section—the ribbon takes the base WPF control class and derives more specialized versions. From a purist point of view, the ToolBar and StatusBar have a cleaner model, because they're able to work with standard WPF controls, which they simply restyle. But the ribbon needs an extra layer of derived classes to support many of its more advanced features. For example, the RibbonApplicationMenu and RibbonApplicationMenuItem are enhanced beyond the ordinary menu classes to support the RibbonCommand.

To create a menu, you create a new RibbonApplicationMenu object and use that to set the Ribbon.ApplicationMenu property. As you probably already expect, the RibbonApplicationMenu includes a collection of RibbonApplicationMenuItem objects, each of which represents a separate clickable menu item.

Here's a basic example outline that creates an application menu with three menu items:

```
<r:Ribbon>
  <r:Ribbon.ApplicationMenu>
    <r:RibbonApplicationMenu>

      <r:RibbonApplicationMenuItem>...</r:RibbonApplicationMenuItem>
      <r:RibbonApplicationMenuItem>...</r:RibbonApplicationMenuItem>
      <r:RibbonApplicationMenuItem>...</r:RibbonApplicationMenuItem>

    </r:RibbonApplicationMenu>
  </r:Ribbon.ApplicationMenu>
</r:Ribbon>
```

As with an ordinary MenuItem, a RibbonApplicationMenuItem needs a value for the Header property (which provides the menu text). However, instead of using the Icon property inherited from MenuItem, you supply a small picture (typically, one that's 32x32 pixels) through the ImageSource property.

Here's an example that fleshes out the three menu items shown earlier. It also adds an image to the small button that opens the menu. (Unlike the pictures next to each command, this button should be 16x16 pixels.)

```
<r:Ribbon>
  <r:Ribbon.ApplicationMenu>
    <r:RibbonApplicationMenu SmallImageSource="images\window2.png">

      <r:RibbonApplicationMenuItem Header="New"
       ToolTip="Create a new document" ImageSource="images\new.png" />
      <r:RibbonApplicationMenuItem Header="Save"
       ToolTip="Save the current document" ImageSource="images\save.png" />
```

```
      <r:RibbonApplicationMenuItem Header="Save As"
       ToolTip="Save the document with a new name"
       ImageSource="images\saveas.png" />

    </r:RibbonApplicationMenu>
  </r:Ribbon.ApplicationMenu>
</r:Ribbon>
```

You handle clicks on the ribbon menu in the same way that you handle clicks in an ordinary menu. You can respond to the Click event, or you can wire up a command using the Command, CommandParameter, and CommandTarget properties. Commands are a particularly good idea with the ribbon menu, because you may want to link the same command to a ribbon menu item and to a ribbon button. They're also required if you want to use the quick access toolbar, as described later.

It's also worth noting that any RibbonApplicationMenuItem can hold more RibbonApplicationMenuItem objects to create a submenu (see Figure 25-9). Each submenu item supports the same text, image, and tooltip options:

```
<r:Ribbon.ApplicationMenu>
  <r:RibbonApplicationMenu SmallImageSource="images\window2.png">
    <r:RibbonApplicationMenuItem Header="New" ImageSource="images\window2.png" />

    <r:RibbonApplicationMenuItem Header="_Save" ImageSource="images\save.png">
      <r:RibbonApplicationMenuItem Header="Save As" ImageSource="images\save.png"/>
      <r:RibbonApplicationMenuItem Header="Save" ImageSource="images\save.png" />
    </r:RibbonApplicationMenuItem>

    <r:RibbonSeparator></r:RibbonSeparator>
    <r:RibbonApplicationMenuItem Header="About" />
    <r:RibbonApplicationMenuItem Header="Exit "/>
  </r:RibbonApplicationMenu>
</r:Ribbon.ApplicationMenu>
```

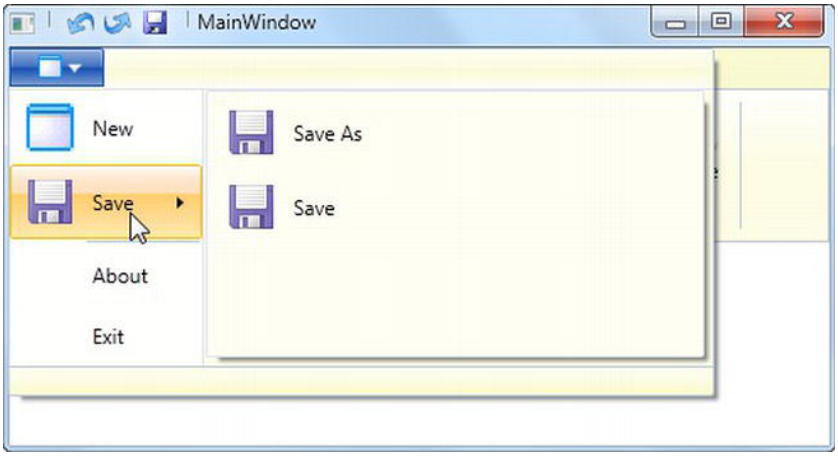


***Figure 25-9.*** *The ribbon application menu with a submenu*

To separate menu items, you can add a thin horizontal dividing line by placing the RibbonSeparator control into your menu. And if you're more ambitious, you can fill the second column of the drop-down menu panel with more information (say, a list of recent documents), and you can add more details to a footer region underneath (for example, a link to a help page). Both regions act like content controls—you simply set the Ribbon.AuxiliaryPaneContent with any element to fill the right-side column, and the Ribbon.FooterPaneContent to fill the footer area. As with any content control, these content properties can hold a layout container with an assortment of interactive elements, or you can supply a data object, which is then interpreted by a template (that you then supply through the AuxiliaryPaneContentTemplate and FooterPaneContent templates).

## Tabs, Groups, and Buttons

The ribbon uses the same model to fill its toolbar tabs as it does to fill its application menu, just with a few extra layers.

First, the ribbon holds a collection of tabs. In turn, each tab holds one or more groups, which is an outlined, titled, box-like section of the ribbon. Lastly, each group holds one or more ribbon controls. Figure 25-10 shows this arrangement.

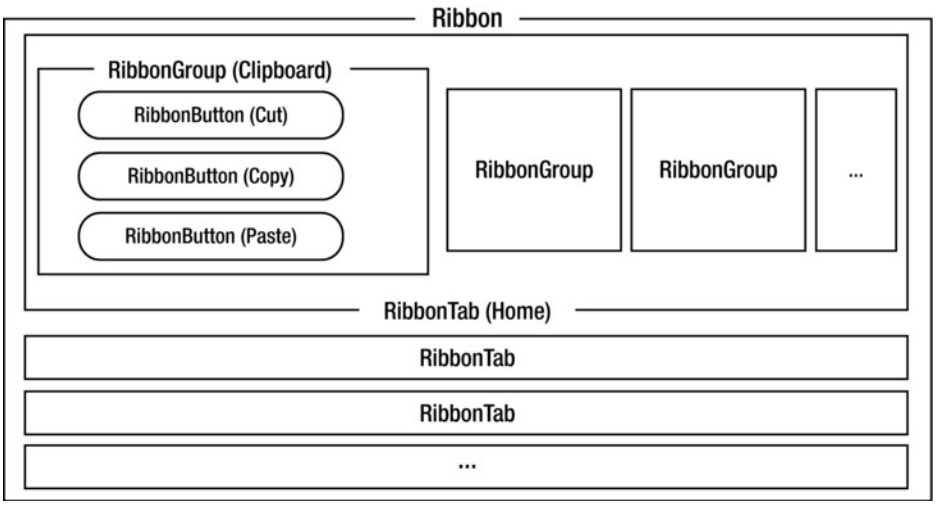


***Figure 25-10.*** *Tabs, groups, and buttons*

Each of these ingredients has a corresponding class. To create a ribbon like the one shown in Figure 25-10, you start by declaring the appropriate RibbonTab objects, fill each one with RibbonGroup objects, and place ribbon controls (like the straightforward RibbonButton) in each group.

When you create a new tab in the ribbon, you use its Header property to supply the text that appears in the tab (above the ribbon). When you create a group in a tab, you use its Header property to supply the text that appears underneath that section of the ribbon. You can also use the SmallImageSource property to set the image that will be used if space is limited and the group is collapsed down to a single button, as shown later, in Figure 25-12.

Here's some markup that gives a ribbon one tab (named "Home"), with one group inside it (named "Clipboard"):

```
<r:Ribbon>
  <r:Ribbon.ApplicationMenu>
    <r:RibbonApplicationMenu>
      ...
    </r:RibbonApplicationMenu>
  </r:Ribbon.ApplicationMenu>

  <r:RibbonTab Header="Home">
    <r:RibbonGroup Header="Clipboard">
      ...
    </r:RibbonGroup>
  </r:RibbonTab>

</r:Ribbon>
```

You can see this part of the ribbon, along with a full set of RibbonCommand buttons, back in Figure 25-9.

As with the application menu, you configure text and image content that appears in each ribbon button. However, the property names are different. Instead of setting Header and ImageSource, you now set Label for the text, and SmallImageSource and LargeImageSource for the image. Two images are needed because the same button can be shown at two different sizes, depending on how you've configured it and how much space is available. You use the familiar Click event or Command property to handle button clicks.

---

■ **Note** The SmallImageSource property sets the image that's used when the item is rendered in small size (16x16 pixels on a standard 96 dpi display). The LargeImageSource property sets the image that's used when the item is rendered in large size (32x32 pixels on a standard 96 dpi display). To avoid scaling artifacts at different pixel densities, you can use a DrawingImage instead of a bitmap for each picture, as explained in Chapter 13.

---

Here's a portion of ribbon markup that defines the Clipboard group (shown in Figure 25-9) and places three commands inside:

```
<r:Ribbon>
  <r:Ribbon.ApplicationMenu>
    <r:RibbonApplicationMenu>
      ...
    </r:RibbonApplicationMenu>
  </r:Ribbon.ApplicationMenu>

  <r:RibbonTab Header="Home">
    <r:RibbonGroup Header="Clipboard">
          <r:RibbonButton Label="Cut"
           SmallImageSource="images/cut.png" LargeImageSource="images/cut.png" />
          <r:RibbonButton Label="Copy"
           SmallImageSource="images/copy.png" LargeImageSource="images/copy.png" />
          <r:RibbonButton Label="Paste"
           SmallImageSource="images/paste.png" LargeImageSource="images/paste.png" />
    </r:RibbonGroup>
  </r:RibbonTab>
```

```
</r:Ribbon>
```

This markup hasn't yet been connected to any logic (either through the Click event or through a command), so clicking these buttons won't trigger any action.

In this example, the ribbon was entirely made up of RibbonButton objects, which is the most common ribbon control type. However, WPF gives you several more options, which are outlined in Table 25-1. As with the application menu, most of the ribbon classes derive from the standard WPF controls. They simply add extra ribbon functionality on top. For example, all of them include the Label property, which lets you add a text caption next to the control (which appears immediately below a big button, to the right of a small button, to the left of a text box or combo box, and so on).

***Table 25-1.*** *Ribbon Control Classes*

| Name | Description |
|---|---|
| RibbonButton | A clickable text-and-image button, which is the most common ingredient on the ribbon. |
| RibbonCheckBox | A check box that can be checked or unchecked. |
| **Name** | **Description** |
| RibbonRadioButton | One clickable option in a group of mutually exclusive options (just like ordinary option buttons). |
| RibbonToggleButton | A button that has two states: pressed or unpressed. For example, many programs use this sort of button to turn on or off font characteristics such as bold, italic, and underline. |
| RibbonMenuButton | A button that pops open a menu. You fill the menu with MenuItem objects using the RibbonMenuButton.Items collection. |
| RibbonSplitButton | Similar to a RibbonMenuButton, but the button is actually divided into two sections. The user can click the top portion (with the picture) to run the command or the bottom portion (with the text and drop-down arrow) to show the linked menu of items. For example, the Paste command in Word is a RibbonSplitButton. |
| RibbonComboBox | Embeds a combo box in the ribbon, which the user can use to type in text or make a selection, just as with the standard ComboBox control. |
| RibbonTextBox | Embeds a text box in the ribbon, which the user can use to type in text, just as with the standard TextBox control. |
| RibbonSeparator | Draws a vertical line between individual controls (or groups of controls) in the ribbon (or a horizontal line between items in a menu). |

## Rich Tooltips

The ribbon supports an enhanced tooltip model, which displays more detailed tooltip pop-ups that can include a title, description, and image (and a footer with the same). However, all these details are optional, and you need only set the ones you want to use.

If you want to use enhanced tooltips, simply remove the standard ToolTip property, and use any combination of the tooltip properties described in Table 25-2. You can set them on any of the ribbon controls, including RibbonButton, and on RibbonApplicationMenuItem objects. The only limitation is that you can't put actual elements (like links) into a tooltip. You're limited to text and image content.

***Table 25-2.*** *Enhanced Properties for ToolTips*

| Property | Description |
|---|---|
| ToolTipTitle | The title that appears at the top of the tooltip for this item. |
| ToolTipDescription | The text that appears in the tooltip, under the title. |
| ToolTipImageSource | The image that appears in the tooltip, under the title and to the left of the text description. The image can be any size. |
| ToolTipFooterTitle | The text that appears a footer title of a tooltip. |
| ToolTipFooterDescription | The text that appears in the footer of a tooltip, under the footer title. |
| ToolTipFooterImageSource | The image that appears to the left of the tooltip footer text. The image can be any size. |

Here's an example of an enhanced tooltip:

```
<r:RibbonButton Label="Cut" ToolTipTitle="Cut"
 ToolTipDescription="Copies the selected text to the clipboard and removes it"
 ToolTipImageSource="images/cut.png"
 ToolTipFooterImageSource="images/help.png"
 ToolTipFooterTitle="More Details" ToolTipFooterDescription="Press F1 for Help" ...
/>
```

Figure 25-11 shows the result.

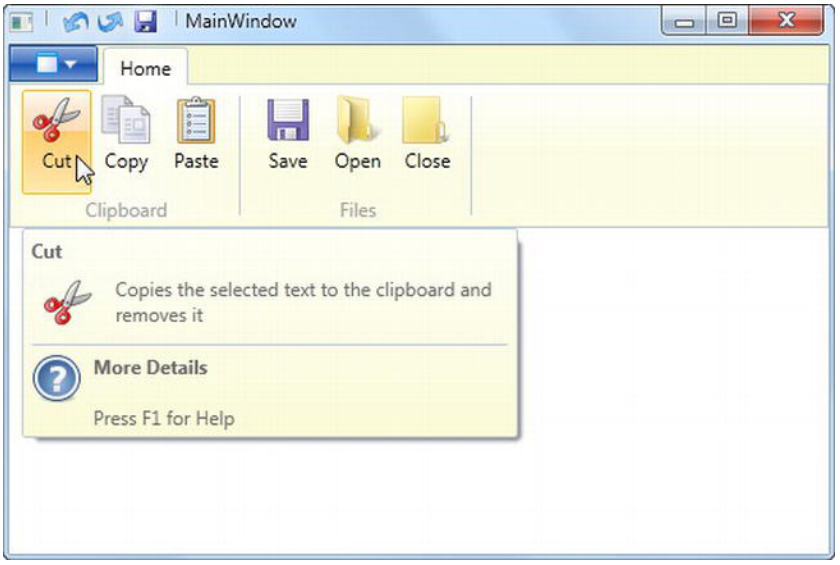


***Figure 25-11.*** *An enhanced tooltip*

## Keyboard Access with KeyTips

Even keyboard users can access the commands in the ribbon. But to do so, you need to assign the appropriate shortcut keys to your tabs, groups, and commands.

Here's how it works if everything's configured properly. First, the user presses the Alt key (and then releases it). The ribbon shows a *keytip*—a single shortcut letter—over the application menu and every tab. The user presses a letter to select a tab (or the application menu), and then the ribbon shows the key tip for each keytip-enabled command in that tab (or menu), as shown in Figure 25-12. Finally, the user presses a letter to trigger the corresponding command. The whole process requires three key presses, and makes it easy for keyboard users to discover the key combination that leads to their command.

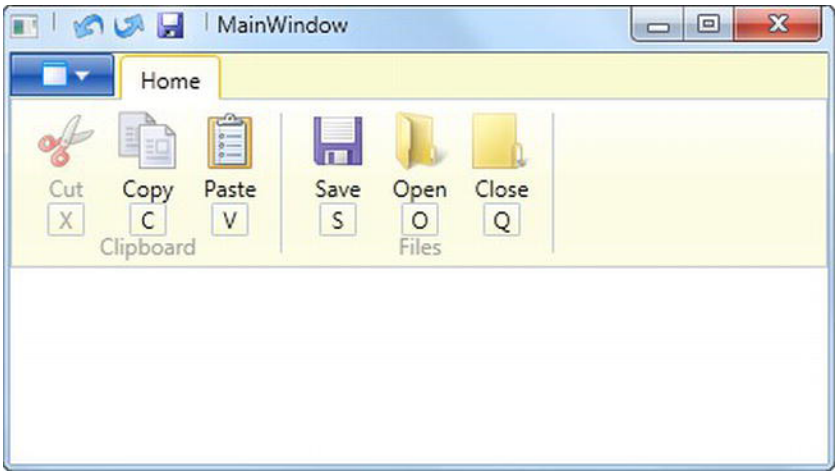


***Figure 25-12.*** *The first level of keytips*

To use keytips in the WPF ribbon, you need to set the KeyTip property on the RibbonApplicationMenu, each RibbonTab, and each RibbonMenuItem, RibbonButton, or other ribbon control. Each keytip must be unique in its scope, which means you shouldn't give two tabs the same keytip letter, or assign the same keytip to two buttons in a tab. And unlike Office applications, you need to keep your keytips to a single letter (two-letter combinations aren't supported). If you don't assign a keytip to a control in the ribbon, it won't be accessible via the keyboard. However, as long as you assign a keytip to the RibbonApplicationMenu, all the menu items inside will be keyboard-accessible, even if they don't have keytips, because the user can use the keytip to open the menu and use the arrow keys to move through the menu.

---

■ **Note** The RibbonMenuItem supports the old-style menu shortcut of using an underscore in the menu text. For example, if you use the menu text "_File" then the F becomes the keytip. However, the same convention doesn't work for commands in the ribbon, so it's probably easiest to always use the KeyTip property and avoid confusion.

---

## Ribbon Sizing

One of the ribbon's most remarkable features is its ability to resize itself to fit the width of the window by reducing and rearranging the buttons in each groups.

When you create a ribbon with WPF, you get basic resizing for free. This resizing, which is built in to the RibbonWrapPanel, uses a different template depending on the number of controls in the group and the size of the group. For example, a group with three RibbonButton objects displays them from left to right, if space permits. If not, the controls on the right are collapsed to small icons, then their text is stripped away to reclaim more space, and finally the whole group is reduced to a single button that, when clicked, shows all the commands in a drop-down list. Figure 25-13 illustrates this process with a ribbon that has three copies of the File group. The first is fully expanded, the second is partially collapsed, and the second is completely collapsed. (It's worth noting that to create this example, the ribbon must be explicitly configured to not collapse the first group. Otherwise, it will always try to partially collapse every group before it fully collapses any group.)

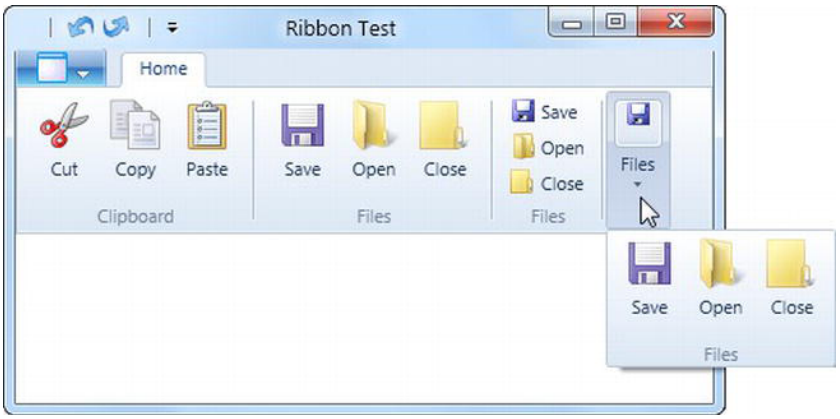


***Figure 25-13.*** *Shrinking the ribbon*

You can use several techniques to change the sizing of a ribbon group. You can use the RibbonTab.GroupSizeReductionOrder property to set which groups should be reduced first. You specify each group using the value of its LabelTitle. Here's an example:

```
<r:RibbonTab Header="Home" GroupSizeReductionOrder="Clipboard,Tasks,File">
```

As you reduce the size of the window, all the groups will be collapsed bit by bit. However, the Clipboard group will switch to a more compact layout first, followed by the Tasks group, and so on. If you continue shrinking the window, there will be another round of group rearrangement, once again led by the Clipboard group. If you don't set the GroupSizeReductionOrder property, the rightmost group leads the way.

A more powerful approach is to create a collection of RibbonGroupSizeDefinition objects that dictates how a group should collapse itself. Each RibbonGroupSizeDefinition is a template that defines a single layout. It specifies which commands should get large icons, which ones should get small icons, and which ones should include display text. Here's an example of a RibbonControlSizeDefinition that sets the layout for a group of four controls, making them all as big as can be:

```
<r:RibbonGroupSizeDefinition>
  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True" />
  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True" />
  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True" />
  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True" />
</r:RibbonGroupSizeDefinition>
```

To understand this, you need to realize that a RibbonGroupSizeDefinition matches controls by order, regardless of their names. So the definition shown provides instructions for the first four controls in the ribbon, regardless of what type of control they are and what text they contain.

To take control of group resizing, you need to define multiple RibbonGroupSizeDefinition objects and order them from largest to smallest in a RibbonGroupSizeDefinitionCollection. As the group is collapsed, the ribbon can then switch from one layout to the next to reclaim more space, while keeping the layout you want (and ensuring that the controls you think are most important remain visible). Usually, you'll place the RibbonGroupSizeDefinitionCollection in the Ribbon.Resources section, so you can reuse the same sequences of templates for more than one four-button group.

```
<r:Ribbon.Resources>
  <r:RibbonGroupSizeDefinitionBaseCollection x:Key="RibbonLayout">

    <!-- All large controls. -->
    <r:RibbonGroupSizeDefinition>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
    </r:RibbonGroupSizeDefinition>

    <!-- A large control at both ends, with two small controls in between. -->
    <r:RibbonGroupSizeDefinition>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
    </r:RibbonGroupSizeDefinition>

    <!-- Same as before, but now with no text for the small buttons. -->
    <r:RibbonGroupSizeDefinition>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
    </r:RibbonGroupSizeDefinition>

    <!-- All small buttons. -->
    <r:RibbonGroupSizeDefinition>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
    </r:RibbonGroupSizeDefinition>
```

```
    <!-- All small, no-text buttons. -->
    <r:RibbonGroupSizeDefinition>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
    </r:RibbonGroupSizeDefinition>

    <!-- Collapse the entire group to a single drop-down button. -->
    <r:RibbonGroupSizeDefinition IsCollapsed="True" />
  </r:RibbonGroupSizeDefinitionBaseCollection>
</r:Ribbon.Resources>
```

Now you can apply these resizing rules to a group in your ribbon like this:

```
<r:RibbonGroup Header="Files" SmallImageSource="images/save_Small.png"
 GroupSizeDefinitions="{StaticResource RibbonLayout}">
  ...
</r:RibbonGroup>
```

---

■ **Note** Not only can the ribbon be resized, it can also be minimized (collapsed down so that just the tabs are visible). Users can minimize the ribbon (and expand it back) by double-clicking any tab title, or by right-clicking the ribbon and choosing Minimize the Ribbon.

---

## The Quick Access Toolbar

The final ingredient that you'll consider in the ribbon is the quick access toolbar (or QAT). It's a narrow strip of commonly used buttons that sits either just above or just below the rest of the ribbon, depending on user selection.

Initially, your ribbon won't include a quick access toolbar. If you want to add one, you need to create it by setting the Ribbon.QuickAccessToolBar property. It takes a RibbonQuickAccessToolbar object, which holds a series of RibbonButton objects. When defining the RibbonCommand for these objects, you need only supply the tooltip text and small image, because text labels and large images are never shown.

Here's the definition for the exceedingly simple QAT shown back in Figure 25-8:

```
<r:Ribbon.QuickAccessToolBar>
  <r:RibbonQuickAccessToolBar>
    <r:RibbonButton Label="Undo" SmallImageSource="images\undo.png" />
    <r:RibbonButton Label="Redo" SmallImageSource="images\redo.png" />
    <r:RibbonButton Label="Save" SmallImageSource="images\save_small.png" />
  </r:RibbonQuickAccessToolBar>
</r:Ribbon.QuickAccessToolBar>
```

The real goal of the QAT is to provide a customization avenue for the user. You should put a relatively small number of items into your QAT, but allow users to take their favorite commands from the ribbon and place them into the QAT. This happens pretty much automatically, with two caveats. Most importantly, a button can only be copied into the QAT if it uses a command (in other words, you've set its Command

property, and you're using that rather than the Click event to trigger the appropriate action in your application). If you've done that, a user can copy any command into the QAT by right-clicking the ribbon button and choosing Add to Quick Access Toolbar (see Figure 25-14). Similarly, a user can remove a button from the QAT by right-clicking it and choosing Remove from Quick Access Toolbar. You can turn this feature off for specific buttons by setting the CanAddToQuickAccessToolbarDirectly property to false.

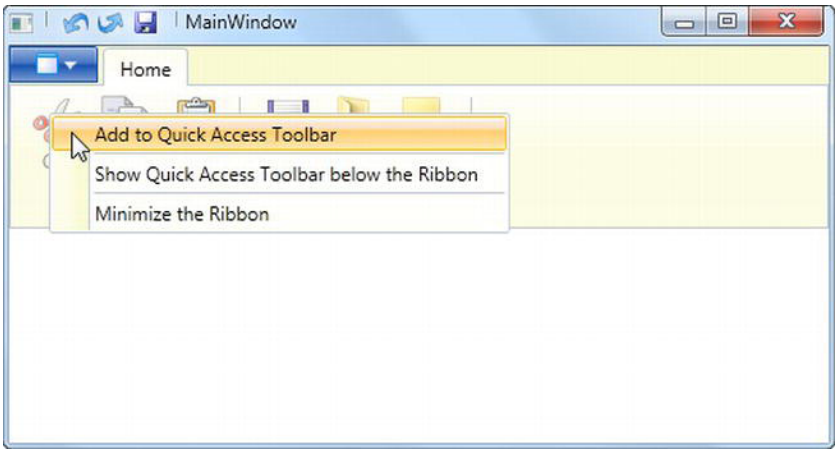


***Figure 25-14.*** *Adding a command to the quick access toolbar*

---

■ **Tip** It's also possible to add items from the application menu, provided they are wired to commands, not Click event handlers. In this case, you'll run into a small issue—the standard menu image, as set by the ImageSource property, is too big for the QAT. The ribbon scales it down automatically, but this can reduce the quality of the image. To avoid this quirk, give each menu command a QAT-compatible picture by adding the QuickAccessToolBarImageSource property.

---

If the ribbon is stuffed full of items, or the window has been resized to very narrow dimension, some of the items in the QAT will be moved into the overflow menu. To see them, the user must click the drop-down arrow at the right edge of the QAT, which opens a drop-down that shows all the extra commands that don't fit.

The most significant limitation of the QAT is that it cannot save its current state. In other words, there's no way for your application to remember the commands that the user has added and restore them the next time the application is launched (unless you build that functionality yourself).

---

■ **Tip** There are still several more ribbon features that aren't covered in this chapter, such as ribbon galleries, custom resizing with a custom layout panel, and contextual tabs. For more information, you can refer to Microsoft's ribbon documentation at `http://tinyurl.com/33yx2cl`.

---

# The Last Word

In this chapter you looked at four controls that underpin professional Windows applications. The first three—the Menu, ToolBar, and StatusBar—derive from the ItemsControl class you considered in Chapter 20. But rather than display data, they hold groups of menu commands, toolbar buttons, and status items. This is one more example that shows how the WPF library takes fundamental concepts, such as the ItemsControl, and uses them to standardize entire branches of the control family.

The fourth and final control that you considered is the Ribbon, a toolbar replacement that was introduced as the distinguishing feature of Office 2007 and became a standard ingredient Windows 7. Although the ribbon isn't baked into the .NET runtime, it's a valuable free library for WPF developers.