CHAPTER 9

Commands

In Chapter 5, you learned about routed events, which you can use to respond to a wide range of mouse and keyboard actions. However, events are a fairly low-level ingredient. In a realistic application, functionality is divided into higher-level *tasks*. These tasks may be triggered by a variety of actions and through a variety of user-interface elements, including main menus, context menus, keyboard shortcuts, and toolbars.

WPF allows you to define these tasks—known as *commands*—and connect controls to them so you don't need to write repetitive event-handling code. Even more important, the command feature manages the state of your user interface by automatically disabling controls when the linked commands aren't available. It also gives you a central place to store (and localize) the text captions for your commands.

In this chapter, you'll learn how to use the prebuilt command classes in WPF, wire them up to controls, and define your own commands. You'll also consider the limitations of the command model—namely, the lack of a command history and the lack of support for an application-wide Undo feature—and you'll see how you can build your own system for tracking and reversing commands.

Understanding Commands

In a well-designed Windows application, the application logic doesn't sit in the event handlers but is coded in higher-level methods. Each one of these methods represents a single application *task*. Each task may rely on other libraries (such as separately compiled components that encapsulate business logic or database access). Figure 9-1 shows this relationship.

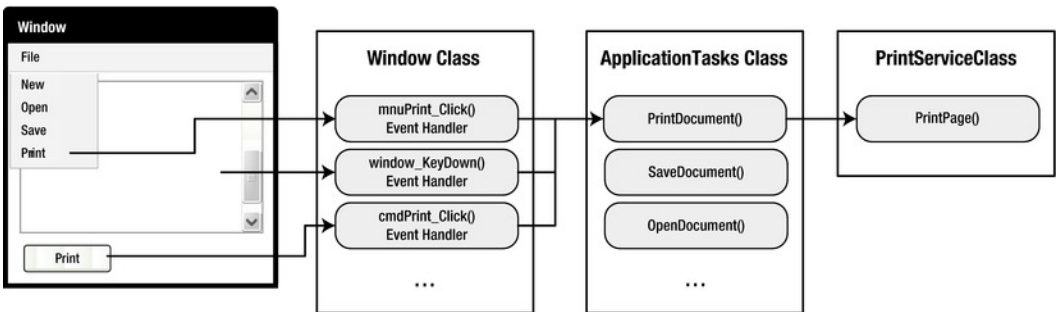

Figure 9-1. *Mapping event handlers to a task*

The most obvious way to use this design is to add event handlers wherever they're needed, and use each event handler to call the appropriate application method. In essence, your window code becomes a stripped-down switchboard that responds to input and forwards requests to the heart of the application.

Although this design is perfectly reasonable, it doesn't save you any work. Many application tasks can be triggered through a variety of routes, so you'll often need to code several event handlers that call the same application method. This in itself isn't much of a problem (because the switchboard code is so simple), but life becomes much more complicated when you need to deal with user interface *state*.

A simple example shows the problem. Imagine you have a program that includes an application method named PrintDocument(). This method can be triggered in four ways: through a main menu (by choosing File ä Print), through a context menu (by right-clicking somewhere and choosing Print), through a keyboard shortcut (Ctrl+P), and through a toolbar button. At certain points in your application's lifetime, you need to temporarily disable the PrintDocument() task. That means you need to disable the two menu commands and the toolbar button so they can't be clicked, and you need to ignore the Ctrl+P shortcut. Writing the code that does this (and adding the code that enables these controls later) is messy. Even worse, if it's not done properly, you might wind up with different blocks of state code overlapping incorrectly, causing a control to be switched on even when it shouldn't be available. Writing and debugging this sort of code is one of the least glamorous aspects of Windows development.

WPF includes a command model that can help you deal with these issues. It adds two key features:

- It delegates events to the appropriate commands.
- It keeps the enabled state of a control synchronized with the state of the corresponding command.

The WPF command model isn't quite as straightforward as you might expect. Plugging into the routed event model requires several separate ingredients, which you'll learn about in this chapter. However, the command model is *conceptually* simple. Figure 9-2 shows how a command-based application changes the design shown in Figure 9-1. Now each action that initiates printing (clicking the button, clicking the menu item, or pressing Ctrl+P) is mapped to the same command. A command binding links that command to a single event handler in your code.

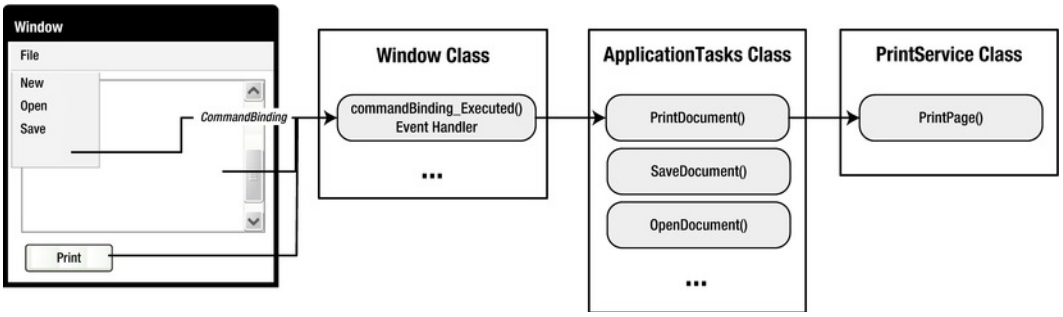

Figure 9-2. *Mapping events to a command*

The WPF command system is a great tool for simplifying application design. However, it still has some fairly significant gaps. Notably, WPF doesn't have any support for the following:

- Command tracking (for example, keeping a history of recent commands)
- Undoable commands

- Commands that have state and can be in different modes (for example, a command that can be toggled on or off)

The WPF Command Model

The WPF command model consists of a surprising number of moving parts. All together, it has four key ingredients:

Commands: A command *represents* an application task and keeps track of whether it can be executed. However, commands don't actually contain the code that *performs* the application task.

Command bindings: Each command binding links a command to the related application logic, for a particular area of your user interface. This factored design is important, because a single command might be used in several places in your application and have a different significance in each place. To handle this, you use the same command with different command bindings.

Command sources: A command source triggers a command. For example, a MenuItem and a Button can both be command sources. Clicking them executes the bound command.

Command targets: A command target is the element on which the command is being performed. For example, a Paste command might insert text into a TextBox, and an OpenFile command might pop a document into a DocumentViewer. The target may or may not be important, depending on the nature of the command.

In the following sections, you'll dig into the first ingredient: the WPF command.

The ICommand Interface

The heart of the WPF command model is the System.Windows.Input.ICommand interface, which defines how commands work. This interface includes two methods and an event:

```
public interface ICommand
{
    void Execute(object parameter);
    bool CanExecute(object parameter);

    event EventHandler CanExecuteChanged;
}
```

In a simple implementation, the Execute() method would contain the application task logic (for example, printing the document). However, as you'll see in the next section, WPF is a bit more elaborate. It uses the Execute() method to fire off a more complicated process that eventually raises an event that's handled elsewhere in your application. This gives you the ability to use ready-made command classes and plug in your own logic. It also gives you the flexibility to use one command (such as Print) in several places.

The CanExecute() method returns the state of the command: true if it's enabled and false if it's disabled. Both Execute() and CanExecute() accept an additional parameter object that you can use to pass along any extra information you need.

Finally, the CanExecuteChanged event is raised when the state changes. This is a signal to any controls using the command that they should call the CanExecute() method to check the command's state. This is part of the glue that allows command sources (such as a Button or MenuItem) to automatically enable themselves when the command is available and to disable themselves when it's not available.

The RoutedCommand Class

When creating your own commands, you won't implement ICommand directly. Instead, you'll use the System.Windows.Input.RoutedCommand class, which implements this interface for you. The RoutedCommand class is the only class in WPF that implements ICommand. In other words, all WPF commands are instances of RoutedCommand (or a derived class).

One of the key concepts behind the command model in WPF is that the RoutedUICommand class doesn't contain any application logic. It simply *represents* a command. This means one RoutedCommand object has the same capabilities as another.

The RoutedCommand class adds a fair bit of extra infrastructure for event tunneling and bubbling. Whereas the ICommand interface encapsulates the idea of a command—an action that can be triggered and may or may not be enabled—the RoutedCommand modifies the command so that it can bubble through the WPF element hierarchy to get to the correct event handler.

WHY WPF COMMANDS NEED EVENT BUBBLING

When looking at the WPF command model for the first time, it's tricky to grasp exactly why WPF commands require routed events. After all, shouldn't the command object take care of performing the command, regardless of how it's invoked?

If you were using the ICommand interface directly to create your own command classes, this would be true. The code would be hardwired into the command, so it would work the same way no matter what triggered the command. You wouldn't need event bubbling.

However, WPF uses a number of *prebuilt* commands. These command classes don't contain any real code. They're just conveniently defined objects that represent a common application task (such as printing a document). To act on these commands, you need to use a command binding, which raises an event to your code (as shown in Figure 9-2). To make sure you can handle this event in one place, even if it's fired by different command sources in the same window, you need the power of event bubbling.

This raises an interesting question: why use prebuilt commands at all? Wouldn't it be clearer to have custom command classes do all the work, instead of relying on an event handler? In many ways, this design would be simpler. However, the advantage of prebuilt commands is that they provide much better possibilities for integration. For example, a third-party developer could create a document viewer control that uses the prebuilt Print command. As long as your application uses the same prebuilt command, you won't need to do any extra work to wire up printing in your application. Seen this way, commands are a major piece of WPF's pluggable architecture.

To support routed events, the RoutedCommand class implements the ICommand interface privately and then adds slightly different versions of its methods. The most obvious change you'll notice is that the Execute() and CanExecute() methods take an extra parameter. Here are their new signatures:

```
public void Execute(object parameter, IInputElement target)
{...}

public bool CanExecute(object parameter, IInputElement target)
{...}
```

The *target* is the element where the event handling begins. This event begins at the target element and bubbles up to higher-level containers until your application handles it to perform the appropriate task. (To handle the Executed event, your element needs the help of yet another class—CommandBinding.)

Along with this shift, the RoutedElement also introduces three properties: the command name (Name), the class that this command is a member of (OwnerType), and any keystrokes or mouse actions that can also be used to trigger the command (in the InputGestures collection).

The RoutedUICommand Class

Most of the commands you'll deal with won't be RoutedCommand objects; rather, they will be instances of the RoutedUICommand class, which derives from RoutedCommand. (In fact, all the ready-made commands that WPF provides are RoutedUICommand objects.)

RoutedUICommand is intended for commands with text that is displayed somewhere in the user interface (for example, the text of a menu item or the tooltip for a toolbar button). The RoutedUICommand class adds a single property—Text—which is the display text for that command.

The advantage of defining the command text with the command (rather than directly on the control) is that you can perform your localization in one place. However, if your command text never appears anywhere in the user interface, the RoutedCommand class is equivalent.

■ **Note** You don't need to use the RoutedUICommand text in your user interface. In fact, there may be good reasons to use something else. For example, you might prefer *Print Document* to just *Print*, and in some cases you might replace the text altogether with a tiny graphic.

The Command Library

The designers of WPF realized that every application is likely to have a large number of commands and that many commands are common to many different applications. For example, all document-based applications will have their own versions of the New, Open, and Save commands. To save you the work of creating those commands, WPF includes a basic command library that's stocked with more than 100 commands. These commands are exposed through the static properties of five dedicated static classes:

ApplicationCommands: This class provides the common commands, including clipboard commands (such as Copy, Cut, and Paste) and document commands (such as New, Open, Save, SaveAs, Print, and so on).

NavigationCommands: This class provides commands used for navigation, including some that are designed for page-based applications (such as BrowseBack, BrowseForward, and NextPage) and others that are suitable for document-based applications (such as IncreaseZoom and Refresh).

EditingCommands: This class provides a long list of mostly document-editing commands, including commands for moving around (MoveToLineEnd, MoveLeftByWord, MoveUpByPage, and so on), selecting content (SelectToLineEnd, SelectLeftByWord), and changing formatting (ToggleBold and ToggleUnderline).

ComponentCommands: This includes commands that are used by user-interface components, including commands for moving around and selecting content that are similar to (and even duplicate) some of the commands in the EditingCommands class.

MediaCommands: This class includes a set of commands for dealing with multimedia (such as Play, Pause, NextTrack, and IncreaseVolume).

The ApplicationCommands class exposes a set of basic commands that are commonly used in all types of applications, so it's worth a quick look. Here's the full list:

New	Copy	SelectAll
Open	Cut	Stop
Save	Paste	ContextMenu
SaveAs	Delete	CorrectionList
Close	Undo	Properties
Print	Redo	Help
PrintPreview	Find	
CancelPrint	Replace	

For example, ApplicationCommands.Open is a static property that exposes a RoutedUICommand object. This object represents the Open command in an application. Because ApplicationCommands.Open is a static property, there is only one instance of the Open command for your entire application. However, you may treat it differently depending on its source—in other words, where it occurs in the user interface.

The RoutedUICommand.Text property for every command matches its name, with the addition of spaces between words. For example, the text for the ApplicationCommands.SelectAll command is *Select All*. (The Name property gives you the same text without the spaces.) The RoutedUICommand.OwnerType property returns a type object for the ApplicationCommands class, because the Open command is a static property of that class.

■ **Tip** You can modify the Text property of a command before you bind it in a window (for example, using code in the constructor of your window or application class). Because commands are static objects that are global to your entire application, changing the text affects the command everywhere it appears in your user interface. Unlike the Text property, the Name property cannot be modified.

As you've already learned, these individual command objects are just markers with no real functionality. However, many of the command objects have one extra feature: default input bindings. For example, the ApplicationCommands.Open command is mapped to the keystroke Ctrl+O. As soon as you

bind that command to a command source and add that command source to a window, the key combination becomes active, even if the command doesn't appear anywhere in the user interface.

Executing Commands

So far, you've taken a close look at commands, considering both the base classes and interfaces and the command library that WPF provides for you to use. However, you haven't yet seen any examples of how to use these commands.

As explained earlier, the RoutedUICommand doesn't have any hardwired functionality. It simply represents a command. To trigger this command, you need a command *source* (or you can use code). To respond to this command, you need a command *binding* that forwards execution to an ordinary event handler. You'll see both ingredients in the following sections.

Command Sources

The commands in the command library are always available. The easiest way to trigger them is to hook them up to a control that implements the ICommandSource interface, which includes controls that derive from ButtonBase (Button, CheckBox, and so on), individual ListBoxItem objects, the Hyperlink, and the MenuItem.

The ICommandSource interface defines three properties, as listed in Table 9-1.

Table 9-1. *Properties of the ICommandSource Interface*

Name	Description
Command	Points to the linked command. This is the only required detail.
CommandParameter	Supplies any other data you want to send with the command.
CommandTarget	Identifies the element on which the command is being performed.

For example, here's a button that links to the ApplicationCommands.New command by using the Command property:

```
<Button Command="ApplicationCommands.New">New</Button>
```

WPF is intelligent enough to search all five command container classes described earlier, which means you can use the following shortcut:

```
<Button Command="New">New</Button>
```

However, you may find that this syntax is less explicit and therefore less clear because it doesn't indicate which class contains the command.

Command Bindings

When you attach a command to a command source, you'll see something interesting. The command source will be automatically disabled.

For example, if you create the New button shown in the previous section, the button will appear dimmed and won't be clickable, just as if you had set IsEnabled to false (see Figure 9-3). That's because the button has queried the state of the command. Because the command has no attached binding, it's considered to be disabled.

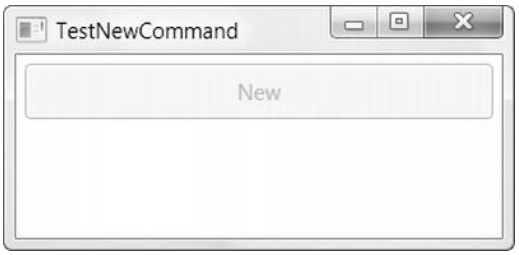

Figure 9-3. *A command without a binding*

To change this state of affairs, you need to create a binding for your command that indicates three things:

- What to do when the command is triggered.
- How to determine whether the command can be performed. (This is optional. If you leave out this detail, the command is always enabled as long as there is an attached event handler.)
- Where the command is in effect. For example, the command might be limited to a single button, or it might be enabled over the entire window (which is more common).

Here's a snippet of code that creates a binding for the New command. You can add this code to the constructor of your window:

```
// Create the binding.
CommandBinding binding = new CommandBinding(ApplicationCommands.New);

// Attach the event handler.
binding.Executed += NewCommand_Executed;

// Register the binding.
this.CommandBindings.Add(binding);
```

Notice that the completed CommandBinding object is added to the CommandBindings collection of the containing window. This works through event bubbling. Essentially, when the button is clicked, the CommandBinding.Executed event bubbles up from the button to the containing elements.

Although it's customary to add all the bindings to the window, the CommandBindings property is actually defined in the base UIElement class. That means it's supported by any element. For example, this example would work just as well if you added the command binding directly to the button that uses it (although then you wouldn't be able to reuse it with another higher-level element). For greatest flexibility, command bindings are usually added to the top-level window. If you want to use the same command from more than one window, you'll need to create a binding in both windows.

■ **Note** You can also handle the CommandBinding.PreviewExecuted event, which is fired first in the highest-level container (the window) and then tunnels down to the button. As you learned in Chapter 4, you use event tunneling to intercept and stop an event before it's completed. If you set the RoutedEventArgs.Handled property to true, the Executed event will never take place.

The previous code assumes that you have an event handler named NewCommand_Executed in the same class, which is ready to receive the command. Here's an example of some simple code that displays the source of the command:

```
private void NewCommand_Executed(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("New command triggered by " + e.Source.ToString());
}
```

Now, when you run the application, the button is enabled (see Figure 9-4). If you click it, the Executed event fires, bubbles up to the window, and is handled by the NewCommand() handler shown earlier. At this point, WPF tells you the source of the event (the button). The ExecutedRoutedEventArgs object also allows you to get a reference to the command that was invoked (ExecutedRoutedEventArgs.Command) and any extra information that was passed along (ExecutedRoutedEventArgs.Parameter). In this example, the parameter is null because you haven't passed any extra information. (If you wanted to pass additional information, you would set the CommandParameter property of the command source. And if you wanted to pass a piece of information drawn from another control, you would need to set CommandParameter by using a data-binding expression, as shown later in this chapter.)

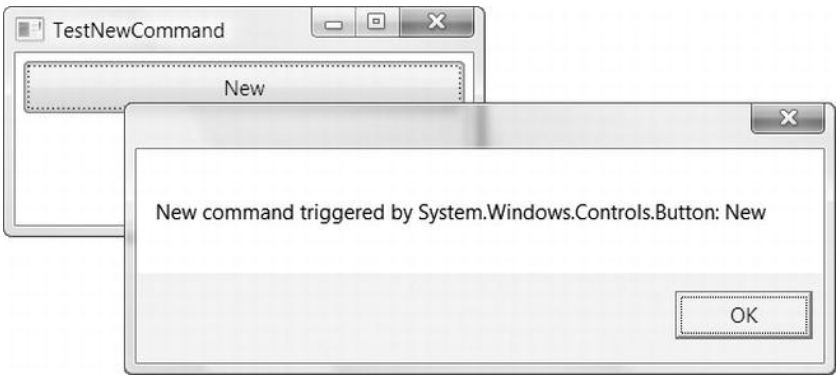

Figure 9-4. *A command with a binding*

■ **Note** In this example, the event handler that responds to the command is still code inside the window where the command originates. The same rules of good code organization still apply to this example; in other words, your window should delegate its work to other components where appropriate. For example, if your command involves opening a file, you may use a custom file helper class that you've created to serialize and deserialize information. Similarly, if you create a command that refreshes a data display, you'll use it to call a method in a database component that fetches the data you need. See Figure 9-2 for a refresher.

In the previous example, the command binding was generated using code. However, it's just as easy to wire up commands declaratively using XAML if you want to streamline your code-behind file. Here's the markup you need:

```
<Window x:Class="Commands.TestNewCommand"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TestNewCommand">
  <Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.New"
      Executed="NewCommand_Executed"></CommandBinding>
  </Window.CommandBindings>

  <StackPanel Margin="5">
    <Button Padding="5" Command="ApplicationCommands.New">New</Button>
  </StackPanel>
</Window>
```

Unfortunately, Visual Studio does not have any design-time support for defining command bindings. It also provides relatively feeble support for connecting controls and commands. You can set the Command property of a control by using the Properties window, but it's up to you to type the exact name of the command—there's no handy drop-down list of commands from which to choose.

Using Multiple Command Sources

The button example seems like a somewhat roundabout way to trigger an ordinary event. However, the extra command layer starts to make more sense when you add more controls that use the same command. For example, you might add a menu item that also uses the New command:

```
<Menu>
  <MenuItem Header="File">
    <MenuItem Command="New"></MenuItem>
  </MenuItem>
</Menu>
```

Note that this MenuItem object for the New command doesn't set the Header property. That's because the MenuItem is intelligent enough to pull the text out of the command if the Header property isn't set. (The Button control lacks this feature.) This might seem like a minor convenience, but it's an important consideration if you plan to localize your application in different languages. In this case, being able to modify the text in one place (by setting the Text property of your commands) is easier than tracking it down in your windows.

The MenuItem class has another frill. It automatically picks up the first shortcut key that's in the Command.InputBindings collection (if there is one). In the case of the ApplicationsCommands.New command object, that means the Ctrl+O shortcut appears in the menu alongside the menu text (see Figure 9-5).

■ **Note** One frill you *don't* get is an underlined access key. WPF has no way of knowing what commands you might place together in a menu, so it can't determine the best access keys to use. This means if you want to use the N key as a quick access key (so that it appears underlined when the menu is opened with the keyboard, and the user can trigger the New command by pressing N), you need to set the menu text manually, preceding the access key with an underscore. The same is true if you want to use a quick access key for a button.

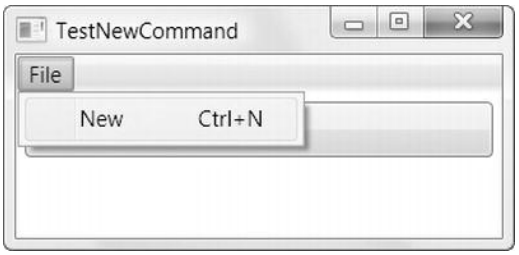

Figure 9-5. *A menu item that uses a command*

Note that you don't need to create another command binding for the menu item. The single command binding you created in the previous section is now being used by two different controls, both of which hand their work off to the same command event handler.

Fine-Tuning Command Text

Based on the ability of the menu to pull out the text of the command item automatically, you might wonder whether you can do the same with other ICommandSource classes, such as the Button control. You can, but it requires a bit of extra work.

You can use two techniques to reuse the command text. One option is to pull the text directly from the static command object. XAML allows you to do this with the Static markup extension. Here's an example that gets the command name New and uses that as the text for a button:

```
<Button Command="New" Content="{x:Static ApplicationCommands.New}"></Button>
```

The problem with this approach is that it simply calls ToString() on the command object. As a result, you get the command name but not the command text. (For commands that have multiple words, the command text is nicer because it includes spaces.) You could correct this problem, but it's significantly more work. There's also another issue in the way that one button uses the same command twice, introducing the possibility that you'll inadvertently grab the text from the wrong command.

The preferred solution is to use a data-binding expression. This data binding is a bit unusual, because it binds to the current element, grabs the Command object you're using, and pulls out the Text property. Here's the terribly long-winded syntax:

```
<Button Margin="5" Padding="5" Command="ApplicationCommands.New" Content=
  "{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"
</Button>
```

You can use this technique in other, more imaginative ways. For example, you can set the content of a button with a tiny image but use the binding expression to show the command name in a tooltip:

```
<Button Margin="5" Padding="5" Command="ApplicationCommands.New"
  ToolTip="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}">
<Image ... />
</Button>
```

The content of the button (which isn't shown here) will be a shape or bitmap that appears as a thumbnail icon.

Clearly, this approach is wordier than just putting the command text directly in your markup. However, it's worth considering if you are planning to localize your application in different languages. You simply need to set the command text for all your commands when your application starts. (If you change

the command text after you've created a command binding, it won't have any effect. That's because the Text property isn't a dependency property, so there's no automatic change notification to update the user interface.)

Invoking a Command Directly

You aren't limited to the classes that implement ICommandSource if you want to trigger a command. You can also call a method directly from any event handler by using the Execute() method. At that point, you need to pass in the parameter value (or a null reference) and a reference to the target element:

```
ApplicationCommands.New.Execute(null, targetElement);
```

The target element is simply the element where WPF begins looking for the command binding. You can use the containing window (which has the command binding) or a nested element (such as the actual element that fired the event).

You can also go through the Execute() method in the associated CommandBinding object. In this case, you don't need to supply the target element, because it's automatically set to the element that exposes the CommandBindings collection that you're using.

```
this.CommandBindings[0].Command.Execute(null);
```

This approach uses only half the command model. It allows you to trigger the command, but it doesn't give you a way to respond to the command's state change. If you want this feature, you may also want to handle the RoutedCommand.CanExecuteChanged to react when the command becomes disabled or enabled. When the CanExecuteChanged event fires, you need to call the RoutedCommand.CanExecute() method to check whether the commands are in a usable state. If not, you can disable or change the content in a portion of your user interface.

COMMAND SUPPORT IN CUSTOM CONTROLS

WPF includes controls that implement ICommandSupport and have the ability to raise commands. (It also includes some controls that have the ability to *handle* commands, as you'll see shortly in the section "Using Controls with Built-in Commands.") Despite this support, you may come across a control that you would like to use with the command model, even though it doesn't implement ICommandSource. In this situation, the easiest option is to handle one of the control's events and execute the appropriate command by using code. However, another option is to build a new control of your own—one that has the command-executing logic built in.

The downloadable code for this chapter includes an example that uses this technique to create a slider that triggers a command when its value changes. This control derives from the Slider class you learned about in Chapter 6; implements ICommand; defines the Command, CommandTarget, and CommandParameter dependency properties; and monitors the RoutedCommand.CanExecuteChanged event internally. Although the code is straightforward, this solution is a bit over the top for most scenarios. Creating a custom control is a fairly significant step in WPF, and most developers prefer to restyle existing controls with templates (discussed in Chapter 17) rather than add an entirely new class. However, if you're designing a custom control from scratch and you want it to provide command support, this example is worth exploring.

Disabling Commands

You'll see the real benefits of the command model when you create a command that varies between an enabled and disabled state. For example, consider the one-window application shown in Figure 9-6, which is a basic text editor that consists of a menu, a toolbar, and a large text box. It allows you to open files, create new (blank) documents, and save your work.

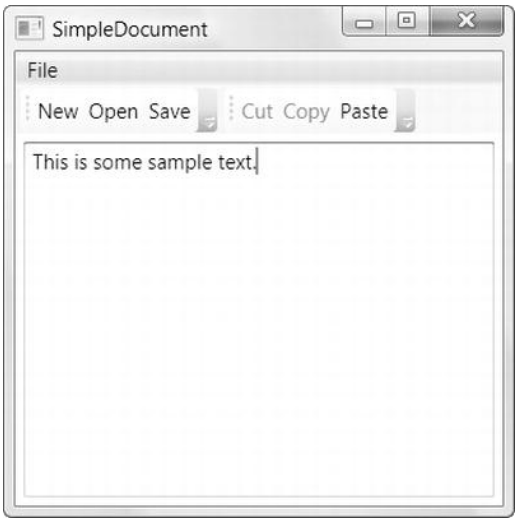

Figure 9-6. *A simple text editor*

In this case, it's perfectly reasonable to make the New, Open, Save, SaveAs, and Close commands perpetually available. But a different design might enable the Save command only if the text has been changed in some way from the original file. By convention, you can track this detail in your code by using a simple Boolean value:

```
private bool isDirty = false;
```

You would then set this flag whenever the text is changed:

```
private void txt_TextChanged(object sender, RoutedEventArgs e)
{
    isDirty = true;
}
```

Now you need a way for the information to make its way from your window to the command binding so that the linked controls can be updated as necessary. The trick is to handle the CanExecute event of the command binding. You can attach an event handler to this event through code:

```
CommandBinding binding = new CommandBinding(ApplicationCommands.Save);
binding.Executed += SaveCommand_Executed;
binding.CanExecute += SaveCommand_CanExecute;
this.CommandBindings.Add(binding);
```

or declaratively:

```
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.Save"
    Executed="SaveCommand_Executed" CanExecute="SaveCommand_CanExecute">
  </CommandBinding>
</Window.CommandBindings>
```

In your event handler, you simply need to check the isDirty variable and set the CanExecuteRoutedEventArg.CanExecute property accordingly:

```
private void SaveCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = isDirty;
}
```

If isDirty is false, the command is disabled. If it's true, the command is enabled. (If you don't set the CanExecute flag, it keeps its most recent value.)

There's one issue to be aware of when using CanExecute. It's up to WPF to call the RoutedCommand.CanExecute() method to trigger your event handler and determine the status of your command. The WPF command manager does this when it detects a change it believes is significant—for example, when the focus moves from one control to another, or after you execute a command. Controls can also raise the CanExecuteChanged event to tell WPF to reevaluate a command—for example, this occurs when you press a key in the text box. All in all, the CanExecute event will fire quite frequently, and you shouldn't use time-consuming code inside it.

However, other factors might affect the command state. In the current example, the isDirty flag could be modified in response to another action. If you notice that the command state is not being updated at the correct time, you can force WPF to call CanExecute() on all the commands you're using. You do this by calling the static CommandManager.InvalidateRequerySuggested() method. The command manager then fires the RequerySuggested event to notify the command sources in your window (buttons, menu items, and so on). The command sources will then requery their linked commands and update themselves accordingly.

THE LIMITS OF WPF COMMANDS

WPF commands are able to change only one aspect of the linked element's state: the value of its IsEnabled property. It's not hard to imagine situations where you need something a bit more sophisticated. For example, you might want to create a PageLayoutView command that can be switched on or off. When switched on, the corresponding controls should be adjusted accordingly. (For example, a linked menu item should be checked (displayed with a checkmark next to it), and a linked toolbar button should be highlighted, as a CheckBox is when you add it to a ToolBar.) Unfortunately, there's no way to keep track of the "checked" state of a command. That means you're forced to handle an event for that control and update its state and that of any other linked controls by hand.

There's no easy way to solve this problem. Even if you created a custom class that derives from RoutedUICommand and gave it the functionality for tracking its checked/unchecked state (and raising an event when this detail changes), you would also need to replace some of the related infrastructure. For example, you would need to create a custom CommandBinding class that could listen to notifications from your custom command, react when the checked/unchecked state changes, and then update the linked controls.

Checked buttons are an obvious example of user-interface state that falls outside the WPF command model. However, other details might suit a similar design. For example, you might create some sort of a split button

that can be switched to different modes. Once again, there's no way to propagate this change to other linked controls through the command model.

Controls with Built-in Commands

Some input controls handle command events on their own. For example, the TextBox class handles the Cut, Copy, and Paste commands (as well as Undo and Redo commands and some of the commands from the EditingCommands class that select text and move the cursor to different positions).

When a control has its own hardwired command logic, you don't need to do anything to make your command work. For example, if you took the simple text editor shown in Figure 9-6 and added the following toolbar buttons, you would get automatic support for cutting, copying, and pasting text.

```
<ToolBar>
  <Button Command="Cut">Cut</Button>
  <Button Command="Copy">Copy</Button>
  <Button Command="Paste">Paste</Button>
</ToolBar>
```

You can click any of these buttons (while the text box has focus) to copy, cut, or paste text from the clipboard. Interestingly, the text box also handles the CanExecute event. If nothing is currently selected in the text box, the Cut and Copy commands will be disabled. All three commands will be automatically disabled when the focus changes to another control that doesn't support these commands (unless you've attached your own CanExecute event handler that enables them).

This example has an interesting detail. The Cut, Copy, and Paste commands are handled by the text box that has focus. However, the command is triggered by the button in the toolbar, which is a completely separate element. In this example, this process works seamlessly because the button is placed in a toolbar, and the ToolBar class includes some built-in magic that dynamically sets the CommandTarget property of its children to the control that currently has focus. (Technically, the ToolBar looks at the parent, which is the window, and finds the most recently focused control in that context, which is the text box. The ToolBar has a separate *focus scope*, and in that context, the button is focused.)

If you place your buttons in a different container (other than a ToolBar or Menu), you won't have this benefit. That means your buttons won't work unless you set the CommandTarget property manually. To do so, you must use a binding expression that names the target element. For example, if the text box is named txtDocument, you would define the buttons like this:

```
<Button Command="Cut"
 CommandTarget="{Binding ElementName=txtDocument}">Cut</Button>
<Button Command="Copy"
 CommandTarget="{Binding ElementName=txtDocument}">Copy</Button>
<Button Command="Paste"
 CommandTarget="{Binding ElementName=txtDocument}">Paste</Button>
```

Another, simpler option is to create a new focus scope by using the attached FocusManager.IsFocusScope property. This tells WPF to look for the element in the parent's focus scope when the command is triggered:

```
<StackPanel FocusManager.IsFocusScope="True">
  <Button Command="Cut">Cut</Button>
  <Button Command="Copy">Copy</Button>
  <Button Command="Paste">Paste</Button>
</StackPanel>
```

This approach has the added advantage that the same commands will apply to multiple controls, unlike the previous example where the CommandTarget was hard-coded. Incidentally, the Menu and ToolBar set the FocusManager.IsFocusScope property to true by default, but you can set it to false if you want the simpler command-routing behavior that doesn't hunt down the focused element in the parent's context.

In some rare cases, you might find that a control has built-in command support you don't want to enable. In this situation, you have three options for disabling the command.

Ideally, the control will provide a property that allows you to gracefully switch off the command support. This ensures that the control will remove the feature and adjust itself consistently. For example, the TextBox control provides an IsUndoEnabled property that you can set to false to prevent the Undo feature. (If IsUndoEnabled is true, the Ctrl+Z keystroke triggers it.)

If that fails, you can add a new binding for the command you want to disable. This binding can then supply a new CanExecute event handler that always responds false. Here's an example that uses this technique to remove support for the Cut feature of the text box:

```
CommandBinding commandBinding = new CommandBinding(
  ApplicationCommands.Cut, null, SuppressCommand);
txt.CommandBindings.Add(commandBinding);
```

and here's the event handler that sets the CanExecute state:

```
private void SuppressCommand(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = false;
    e.Handled = true;
}
```

Notice that this code sets the Handled flag to prevent the text box from performing its own evaluation, which might set CanExecute to true.

This approach isn't perfect. It successfully disables both the Cut keystroke (Ctrl+X) and the Cut command in the context menu for the text box. However, the option will still appear in the context menu in a disabled state.

The final option is to remove the input that triggers the command by using the InputBindings collections. For example, you could disable the Ctrl+C keystroke that triggers the Copy command in a TextBox by using code like this:

```
KeyBinding keyBinding = new KeyBinding(
  ApplicationCommands.NotACommand, Key.C, ModifierKeys.Control);
txt.InputBindings.Add(keyBinding);
```

The trick is to use the special ApplicationCommands.NotACommand value, which is a command that does nothing. It's specifically intended for disabling input bindings.

When you use this approach, the Copy command is still enabled. You can trigger it through buttons of your own creation (or the context menu for the text box, unless you remove that as well by setting the ContextMenu property to null).

■ **Note** You always need to add new command bindings or input bindings to disable features. You can't remove existing bindings. That's because existing bindings don't show up in the public CommandBinding and InputBinding collection. Instead, they're defined through a separate mechanism, called *class bindings*. In Chapter 18, you'll learn how to wire up commands in this way to the custom controls you build.

Advanced Commands

Now that you've seen the basics of commands, it's worth considering a few more sophisticated implementations. In the following sections, you'll learn how to use your own commands, treat the same command differently depending on the target, and use command parameters. You'll also consider how you can support a basic undo feature.

Custom Commands

As well stocked as the five command classes (ApplicationCommands, NavigationCommands, EditingCommands, ComponentCommands, and MediaCommands) are, they obviously can't provide everything your application might need. Fortunately, it's easy to define your own custom commands. All you need to do is instantiate a new RoutedUICommand object.

The RoutedUICommand class provides several constructors. You can create a RoutedUICommand with no additional information, but you'll almost always want to supply the command name, the command text, and the owning type. In addition, you may want to supply a keyboard shortcut for the InputGestures collection.

The best design is to follow the example of the WPF libraries and expose your custom commands through static properties. Here's an example with a command named Requery:

```
public class DataCommands
{
    private static RoutedUICommand requery;

    static DataCommands()
    {
        // Initialize the command.
        InputGestureCollection inputs = new InputGestureCollection();
        inputs.Add(new KeyGesture(Key.R, ModifierKeys.Control, "Ctrl+R"));
        requery = new RoutedUICommand(
          "Requery", "Requery", typeof(DataCommands), inputs);
    }

    public static RoutedUICommand Requery
    {
        get { return requery; }
    }
}
```

■ **Tip** You can also modify the RoutedCommand.InputGestures collection of an existing command—for example, by removing existing key bindings or adding new ones. You can even add mouse bindings, so a command is triggered when a combination of a mouse button and modifier key is pressed (although, in this case, you'll want to place the command binding on just the element where the mouse handling should come into effect).

Once you've defined a command, you can use it in your command bindings in the same way as any of the ready-made commands that are provided by WPF. However, there's one twist. If you want to use your command in XAML, you need to first map your .NET namespace to an XML namespace. For example, if

your class is in a namespace named Commands (the default for a project named Commands), you would add this namespace mapping:

```
xmlns:local="clr-namespace:Commands"
```

In this example, *local* is chosen as the namespace alias. You can use any alias you want, as long as you are consistent in your XAML file.

Now you can access your command through the local namespace:

```
<CommandBinding Command="local:DataCommands.Requery"
  Executed="RequeryCommand_Executed"></CommandBinding>
```

Here's a complete example of a simple window that includes a button that triggers the Requery command:

```
<Window x:Class="Commands.CustomCommand"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CustomCommand" Height="300" Width="300">

  <Window.CommandBindings>
    <CommandBinding Command="local:DataCommands.Requery"
     Executed="RequeryCommand_Executed"></CommandBinding>
  </Window.CommandBindings>

  <Button Margin="5" Command="local:DataCommands.Requery">Requery</Button>
</Window>
```

To complete this example, you simply need to implement the RequeryCommand_Executed() event handler in your code. Optionally, you can also use the CanExecute event to selectively enable or disable this command.

■ **Tip** When using custom commands, you may need to call the static CommandManager.InvalidateRequerySuggested() method to tell WPF to reevaluate the state of your command. WPF will then trigger the CanExecute event and update any command sources that use that command.

Using the Same Command in Different Places

One of the key ideas in the WPF command model is *scope*. Although there is exactly one copy of every command, the effect of using the command varies depending on where it's triggered. For example, if you have two text boxes, they both support the Cut, Copy, and Paste commands, but the operation happens only in the text box that currently has focus.

You haven't yet learned how to do this with the commands that you wire up yourself. For example, imagine you create a window with space for two documents, as shown in Figure 9-7.

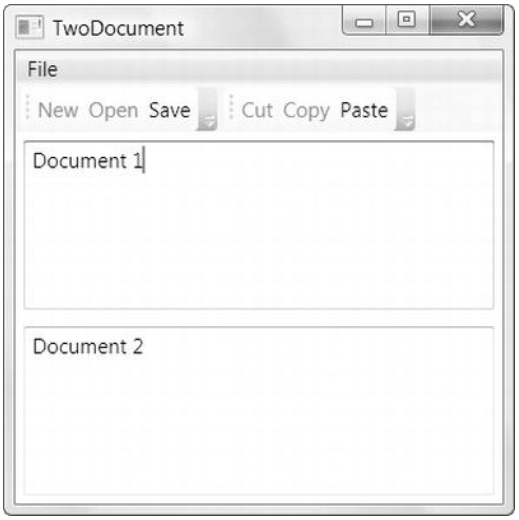

Figure 9-7. *A two-files-at-once text editor*

If you use the Cut, Copy, and Paste commands, you'll find they automatically work on the correct text box. However, the commands you've implemented yourself—New, Open, and Save—do not. The problem is that when the Executed event fires for one of these commands, you have no idea whether it pertains to the first or second text box. Although the ExecutedRoutedEventArgs object provides a Source property, this property reflects the element that has the command binding (just like the sender reference). So far, all your command bindings have been attached to the containing window.

The solution to this problem is to bind the command differently in each text box by using the CommandBindings collection for the text box. Here's an example:

```
<TextBox.CommandBindings>
  <CommandBinding Command="ApplicationCommands.Save"
    Executed="SaveCommand_Executed"
    CanExecute="SaveCommand_CanExecute"></CommandBinding>
</TextBox.CommandBindings>
```

Now the text box handles the Executed event. In your event handler, you can use this information to make sure the correct information is saved:

```
private void SaveCommand_Executed(object sender, ExecutedRoutedEventArgs e)
{
    string text = ((TextBox)sender).Text;
    MessageBox.Show("About to save: " + text);
    ...
    isDirty = false;
}
```

This implementation has two minor issues. First, the simple isDirty flag no longer works, because you must keep track of two text boxes. This problem has several solutions. You could use the TextBox.Tag property to store the isDirty flag. That way, whenever the CanExecuteSave() method is called, you simply look at the Tag property of the sender. Or, you could create a private dictionary collection that stores the

isDirty value, indexed by the control reference. When the CanExecuteSave() method is triggered, you simply look for the isDirty value that belongs to the sender. Here's the full code you would use:

```
private Dictionary<Object, bool> isDirty = new Dictionary<Object, bool>();

private void txt_TextChanged(object sender, RoutedEventArgs e)
{
    isDirty[sender] = true;
}

private void SaveCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    if (isDirty.ContainsKey(sender) && isDirty[sender])
    {
        e.CanExecute = true;
    }
    else
    {
        e.CanExecute = false;
    }
}
```

The other issue with the current implementation is that it creates two command bindings where you really need only one. This adds clutter to your XAML file and makes it more difficult to maintain. This problem is especially bad if you have a large number of commands that are shared between both text boxes.

The solution is to create a single command binding and add that same binding to the CommandBindings collection of both text boxes. This is easy to accomplish in code. If you want to polish it off in XAML, you need to use WPF resources. You simply add a section to the top of your window that creates the CommandBinding object you need to use and gives it a key name:

```
<Window.Resources>
  <CommandBinding x:Key="binding" Command="ApplicationCommands.Save"
    Executed="SaveCommand" CanExecute="CanExecuteSave">
  </CommandBinding>
</Window.Resources>
```

To insert the object into another place in your markup, you use the StaticResource extension and supply the key name:

```
<TextBox.CommandBindings>
  <StaticResource ResourceKey="binding"></StaticResource>
</TextBox.CommandBindings>
```

Using a Command Parameter

So far, the examples you've seen haven't used the command parameter to pass extra information. However, some commands always require some extra information. For example, the NavigationCommands.Zoom command needs a percentage value to use for its zoom. Similarly, you can imagine that some of the commands you're already using might require extra information in certain scenarios. For example, if you use the Save command with the two-file text editor in Figure 9-7, you need to know which file to use when saving the document.

The solution is to set the CommandParameter property. You can set this directly on an ICommandSource control (and you can even use a binding expression that gets a value from another control). For example, here's how you might set the zoom percentage for a button that's linked to the Zoom command by reading the value from another text box:

```
<Button Command="NavigationCommands.Zoom"
  CommandParameter="{Binding ElementName=txtZoom, Path=Text}">
  Zoom To Value
</Button>
```

Unfortunately, that approach doesn't always work. For example, in the two-file text editor, the Save button is reused for each text box, but each text box needs to use a different file name. In situations like these, you're forced to store the information somewhere else (for example, in the TextBox.Tag property or in a separate collection that indexes file names to line up with your text boxes), or you need to trigger the command programmatically like this:

```
ApplicationCommands.New.Execute(theFileName, (Button)sender);
```

Either way, the parameter is made available in the Executed event handler through the ExecutedRoutedEventArgs.Parameter property.

Tracking and Reversing Commands

One feature that the command model lacks is the ability to make a command reversible. Although there is an ApplicationCommands.Undo command, this command is generally used by edit controls (such as the TextBox) that maintain their own Undo histories. If you want to support an application-wide Undo feature, you need to track the previous state internally and restore it when the Undo command is triggered.

Unfortunately, it's not easy to extend the WPF command system. Relatively few entry points are available for you to connect custom logic, and those that exist are not documented. To create a general-purpose, reusable Undo feature, you would need to create a whole new set of "undoable" command classes and a specialized type of command binding. In essence, you would be forced to replace the WPF command system with a new one of your own creation.

A better solution is to design your own system for tracking and reversing commands, but use the CommandManager class to keep a command history. Figure 9-8 shows an example that does exactly that. The window consists of two text boxes, where you can type freely, and a list box that keeps track of every command that has taken place in both text boxes. You can reverse the last command by clicking the Reverse Last Action button.

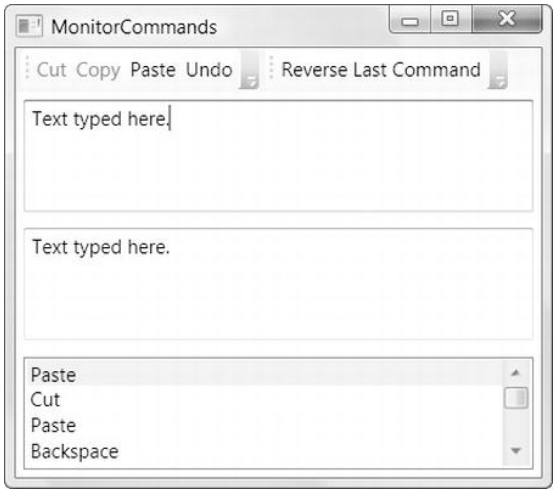

Figure 9-8. *An application-wide Undo feature*

To build this solution, you need to use a few new techniques. The first detail is a class for tracking the command history. It might occur to you to build an undo system that stores a list of recent commands. (Perhaps you would even like to create a derived ReversibleCommand class that exposes a method such as Unexecute() for reversing the task it did previously.) Unfortunately, this system won't work, because all WPF commands are treated like singletons. That means there is only one instance of each command in your application.

To understand the problem, imagine you support the EditingCommands.Backspace command, and the user performs several backspaces in a row. You can register that fact by adding the Backspace command to a stack of recent commands, but you're actually adding the same command object several times. As a result, there's no easy way to store other information along with that command, such as the character that has just been deleted. If you want to store this state, you'll need to build your own data structure to do it. This example uses a class named CommandHistoryItem.

Every CommandHistoryItem object tracks several pieces of information:

- The command name.
- The element on which the command was performed. In this example, there are two text boxes, so it could be either one.
- The property that was changed in the target element. In this example, it will be the Text property of the TextBox class.
- An object that you can use to store the previous state of the affected element (for example, the text the text box had before the command was executed).

■ **Note** This design is fairly crafty in that it stores the state for one element. If you stored a snapshot of the state in the entire window, you would use significantly more memory. However, if you have large amounts of data (such as text boxes with dozens of lines), the Undo overhead could be more than trivial. The solution is to limit the number of items you keep in the history, or to use a more intelligent (and more complex) routine that stores information about only the changed data, rather than *all* the data.

The CommandHistoryItem also includes one method: an all-purpose Undo() method. This method uses reflection to apply the previous value to the modified property. This works for restoring the text in a TextBox, but in a more complex application, you would need a hierarchy of CommandHistoryItem classes, each of which is able to revert a different type of action in a different way.

Here's the complete code for the CommandHistoryItem class, which conserves some space by using the C# language feature *automatic properties*:

```
public class CommandHistoryItem
{
    public string CommandName
    { get; set; }

    public UIElement ElementActedOn
    { get; set; }

    public string PropertyActedOn
    { get; set; }

    public object PreviousState
    { get; set; }

    public CommandHistoryItem(string commandName)
      : this(commandName, null, "", null)
    { }

    public CommandHistoryItem(string commandName, UIElement elementActedOn,
      string propertyActedOn, object previousState)
    {
        CommandName = commandName;
        ElementActedOn = elementActedOn;
        PropertyActedOn = propertyActedOn;
        PreviousState = previousState;
    }

    public bool CanUndo
    {
        get { return (ElementActedOn != null && PropertyActedOn != ""); }    }

    public void Undo()
    {
        Type elementType = ElementActedOn.GetType();
        PropertyInfo property = elementType.GetProperty(PropertyActedOn);
        property.SetValue(ElementActedOn, PreviousState, null);
    }
}
```

The next ingredient you need is a command that performs the application-wide Undo action. The ApplicationCommands.Undo command isn't suitable, because it's already used for individual controls for a different purpose (reverting the last editing change). Instead, you need to create a new command, as shown here:

```
private static RoutedUICommand applicationUndo;

public static RoutedUICommand ApplicationUndo
{
    get { return MonitorCommands.applicationUndo; }
}

static MonitorCommands()
{
    applicationUndo = new RoutedUICommand(
      "ApplicationUndo", "Application Undo", typeof(MonitorCommands));
}
```

In this example, the command is defined in a window class named MonitorCommands.

So far, this code is relatively unremarkable (aside from the nifty bit of reflection code that performs the undo operation). The more difficult part is integrating this command history into the WPF command model. An ideal solution would do this in such a way that you can track any command, regardless of how it's triggered and how it's bound. In a poorly designed solution, you would be forced to rely on a whole new set of custom command objects that have this logic built in or to manually handle the Executed event of every command.

It's easy enough to react to a specific command, but how can you react when *any* command executes? The trick is to use the CommandManager, which exposes a few static events. These events include CanExecute, PreviewCanExecute, Executed, and PreviewCanExecuted. In this example, the last two are the most interesting, because they fire whenever any command is executed.

The Executed event is suppressed by the CommandManager, but you can still attach an event handler by using the UIElement.AddHandler() method and passing in a value of true for the optional third parameter. This allows you to receive the event even though it's handled, as described in Chapter 4. However, the Executed event fires *after* the event is executed, at which point it's too late to save the state of the affected control in your command history. Instead, you need to respond to the PreviewExecuted event, which fires just before.

Here's the code that attaches the PreviewExecuted event handler in the window constructor and removes it when the window is closed:

```
public MonitorCommands()
{
    InitializeComponent();

    this.AddHandler(CommandManager.PreviewExecutedEvent,
      new ExecutedRoutedEventHandler(CommandExecuted));
}

private void window_Unloaded(object sender, RoutedEventArgs e)
{
    this.RemoveHandler(CommandManager.PreviewExecutedEvent,
      new ExecutedRoutedEventHandler(CommandExecuted));
}
```

When the PreviewExecuted event fires, you need to determine whether it's a command that deserves your attention. If so, you can create the CommandHistoryItem and add it to the Undo stack. You also need to watch out for two potential problems. First, when you click a toolbar button to perform a command on the text box, the CommandExecuted event is raised twice: once for the toolbar button and once for the text box. This code avoids duplicate entries in the Undo history by ignoring the command if the sender is

ICommandSource. Second, you need to explicitly ignore the commands you don't want to add to the Undo history. One example is the ApplicationUndo command, which allows you to reverse the previous action.

```
private void CommandExecuted(object sender, ExecutedRoutedEventArgs e)
{
    // Ignore menu button source.
    if (e.Source is ICommandSource) return;

    // Ignore the ApplicationUndo command.
    if (e.Command == MonitorCommands.ApplicationUndo) return;

    TextBox txt = e.Source as TextBox;
    if (txt != null)
    {
        RoutedCommand cmd = (RoutedCommand)e.Command;
        CommandHistoryItem historyItem = new CommandHistoryItem(
          cmd.Name, txt, "Text", txt.Text);

        ListBoxItem item = new ListBoxItem();
        item.Content = historyItem;
        lstHistory.Items.Add(historyItem);
    }
}
```

This example stores all CommandHistoryItem objects in a ListBox. The ListBox has DisplayMember set to Name so that it shows the CommandHistoryItem.Name property of each item. This code supports the Undo feature only if the command is being fired for a text box. However, it's generic enough to work with any text box in the window. You could extend this code to support other controls and properties.

The last detail is the code that performs the application-wide Undo. Using a CanExecute handler, you can make sure that this code is executed only when there is at least one item in the Undo history:

```
private void ApplicationUndoCommand_CanExecute(object sender,
  CanExecuteRoutedEventArgs e)
{
    if (lstHistory == null || lstHistory.Items.Count == 0)
      e.CanExecute = false;
    else
      e.CanExecute = true;
}
```

To revert the last change, you simply call the Undo() method of the CommandHistoryItem and then remove it from the list:

```
private void ApplicationUndoCommand_Executed(object sender, RoutedEventArgs e)
{
    CommandHistoryItem historyItem = (CommandHistoryItem)
      lstHistory.Items[lstHistory.Items.Count - 1];

    if (historyItem.CanUndo) historyItem.Undo();
    lstHistory.Items.Remove(historyItem);
}
```

Although this example demonstrates the concept and presents a simple application with multiple controls that fully support the Undo feature, you would need to make many refinements before you would use an approach like this in a real-world application. For example, you would need to spend considerable time refining the event handler for the CommandManager.PreviewExecuted event to ignore commands that clearly shouldn't be tracked. (Currently, events such as selecting text with the keyboard and hitting the spacebar raise commands.) Similarly, you probably would want to add CommandHistoryItem objects for actions that should be reversible but aren't represented by commands, such as typing a bunch of text and then navigating to another control. Finally, you probably would want to limit the Undo history to just the most recent commands.

The Last Word

In this chapter, you explored the WPF command model. You learned how to hook controls to commands, respond when the commands are triggered, and handle commands differently based on where they occur. You also designed your own custom commands and learned how to extend the WPF command system with a basic command history and Undo feature.

As you've seen in this chapter, the WPF command model isn't quite as streamlined as other bits of WPF architecture. The way that it plugs into the routed event model requires a fairly complex assortment of classes, and its inner workings aren't easy to customize. In fact, to get the most out of WPF commands, you probably need to use a separate toolkit that *extends* WPF by using the Model View ViewModel (MVVM) pattern. The most popular example is Prism (see `http://compositewpf.codeplex.com`).

■ **Tip** Even if you're building huge WPF applications with a team of developers, and Prism interests you, don't delve into it just yet. It's best to learn the fundamentals of WPF first (by reading this book), before you add a whole new layer of complexity.
