Another option is to show the View button only in a selected item. This technique involves modifying or replacing the template you're using in this list, which is described in the "Templates and Selection" section a bit later in this chapter.

Varying Templates

One limitation with the templates you've seen so far is that you're limited to one template for the entire list. But in many situations, you'll want the flexibility to present different data items in different ways.

You can achieve this goal in several ways. Here are some common techniques:

- **Use a data trigger.** You can use a trigger to change a property in the template based on the value of a property in the bound data object. Data triggers work like the property triggers you learned about with styles in Chapter 11, except they don't require dependency properties.
- **Use a value converter.** A class that implements IValueConverter can convert a value from your bound object to a value you can use to set a formatting-related property in your template.
- **Use a template selector.** A template selector examines the bound data object and chooses between several distinct templates.

Data triggers offer the simplest approach. The basic technique is to set a property of one of the elements in your template based on a property in your data item. For example, you could change the background of the custom border that wraps each list item based on the CategoryName property of the corresponding Product object. Here's an example that highlights products in the Tools category with red lettering:

```
<DataTemplate x:Key="DefaultTemplate">
  <DataTemplate.Triggers>
    <DataTrigger Binding="{Binding Path=CategoryName}" Value="Tools">
     <Setter Property="ListBoxItem.Foreground" Value="Red"></Setter>
    </DataTrigger>
  </DataTemplate.Triggers>
  <Border Margin="5" BorderThickness="1" BorderBrush="SteelBlue"
   CornerRadius="4">
    <Grid Margin="3">
      <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
      </Grid.RowDefinitions>
      <TextBlock FontWeight="Bold"
       Text="{Binding Path=ModelNumber}"></TextBlock>
      <TextBlock Grid.Row="1"
       Text="{Binding Path=ModelName}"></TextBlock>
    </Grid>
  </Border>
</DataTemplate>
```

Because the Product object implements the INotifyPropertyChanged interface (as described in Chapter 19), any changes are picked up immediately. For example, if you modify the CategoryName property to move a product out of the Tools category, its text in the list changes at the same time.

This approach is useful but inherently limited. It doesn't allow you to change complex details about your template, only tweak individual properties of the elements in the template (or the container element). Also, as you learned in Chapter 11, triggers can test only for equality—they don't support more complex comparison conditions. That means you can't use this approach to highlight prices that exceed a certain value, for example. And if you need to choose between a range of possibilities (for example, giving each product category a different background color), you'll need to write one trigger for each possible value, which is messy.

Another option is to create one template that's intelligent enough to adjust itself based on the bound object. To pull this trick off, you usually need to use a value converter that examines a property in your bound object and returns a more suitable value. For example, you could create a CategoryToColorConverter that examines a product's category and returns a corresponding Color object. That way, you can bind directly to the CategoryName property in your template, as shown here:

```
<Border Margin="5" BorderThickness="1" BorderBrush="SteelBlue" CornerRadius="4"
 Background=
 "{Binding Path=CategoryName, Converter={StaticResource CategoryToColorConverter}">
```

Like the trigger approach, the value converter approach also prevents you from making dramatic changes, such as replacing a portion of your template with something completely different. However, it allows you to implement more sophisticated formatting logic. Also, it allows you to base a single formatting property on several properties from the bound data object, if you use IMultiValueConverter interface instead of the ordinary IValueConverter.

■ **Tip** Value converters are a good choice if you might want to reuse your formatting logic with other templates.

Template Selectors

Another, more powerful option is to give different items a completely different template. To do this, you need to create a class that derives from DataTemplateSelector. Template selectors work in the same way as the style selectors you considered earlier—they examine the bound object and choose a suitable template using the logic you supply.

Earlier, you saw how to build a style selector that searches for specific values and highlights them with a style. Here's the analogous template selector, which looks at a property (specified by PropertyToEvaluate) and returns the HighlightTemplate if the property matches a set value (specified by PropertyValueToHighlight) or the DefaultTemplate otherwise:

```
public class SingleCriteriaHighlightTemplateSelector : DataTemplateSelector
{
    public DataTemplate DefaultTemplate
    {
        get; set;
    }
```

```
    public DataTemplate HighlightTemplate
    {
        get; set;
    }

    public string PropertyToEvaluate
    {
        get; set;
    }

    public string PropertyValueToHighlight
    {
        get; set;
    }

    public override DataTemplate SelectTemplate(object item,
      DependencyObject container)
    {
        Product product = (Product)item;

        // Use reflection to get the property to check.
        Type type = product.GetType();
        PropertyInfo property = type.GetProperty(PropertyToEvaluate);

        // Decide if this product should be highlighted
        // based on the property value.
        if (property.GetValue(product, null).ToString() == PropertyValueToHighlight)
        {
            return HighlightTemplate;
        }
        else
        {
            return DefaultTemplate;
        }
    }
}
```

And here's the markup that creates the two templates and an instance of the SingleCriteriaHighlightTemplateSelector:

```
<Window.Resources>
  <DataTemplate x:Key="DefaultTemplate">
    <Border Margin="5" BorderThickness="1" BorderBrush="SteelBlue"
      CornerRadius="4">
      <Grid Margin="3">
        <Grid.RowDefinitions>
          <RowDefinition></RowDefinition>
          <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
```

```
        <TextBlock
         Text="{Binding Path=ModelNumber}"></TextBlock>
        <TextBlock Grid.Row="1"
         Text="{Binding Path=ModelName}"></TextBlock>
      </Grid>
    </Border>
  </DataTemplate>

  <DataTemplate x:Key="HighlightTemplate">
    <Border Margin="5" BorderThickness="1" BorderBrush="SteelBlue"
     Background="LightYellow" CornerRadius="4">
      <Grid Margin="3">
        <Grid.RowDefinitions>
          <RowDefinition></RowDefinition>
          <RowDefinition></RowDefinition>
          <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <TextBlock FontWeight="Bold"
         Text="{Binding Path=ModelNumber}"></TextBlock>
        <TextBlock Grid.Row="1" FontWeight="Bold"
         Text="{Binding Path=ModelName}"></TextBlock>
        <TextBlock Grid.Row="2" FontStyle="Italic" HorizontalAlignment="Right">
         *** Great for vacations ***</TextBlock>
      </Grid>
    </Border>
  </DataTemplate>
</Window.Resources>
```

And here's the markup that applies the template selector:

```
<ListBox Name="lstProducts" HorizontalContentAlignment="Stretch">
  <ListBox.ItemTemplateSelector>
    <local:SingleCriteriaHighlightTemplateSelector
      DefaultTemplate="{StaticResource DefaultTemplate}"
      HighlightTemplate="{StaticResource HighlightTemplate}"
      PropertyToEvaluate="CategoryName"
      PropertyValueToHighlight="Travel"
    >
    </local:SingleCriteriaHighlightTemplateSelector>
  </ListBox.ItemTemplateSelector>
</ListBox>
```

As you can see, template selectors are far more powerful than style selectors, because each template has the ability to show different elements arranged in a different layout. In this example, the HighlightTemplate adds a TextBlock with an extra line of text at the end (Figure 20-13).

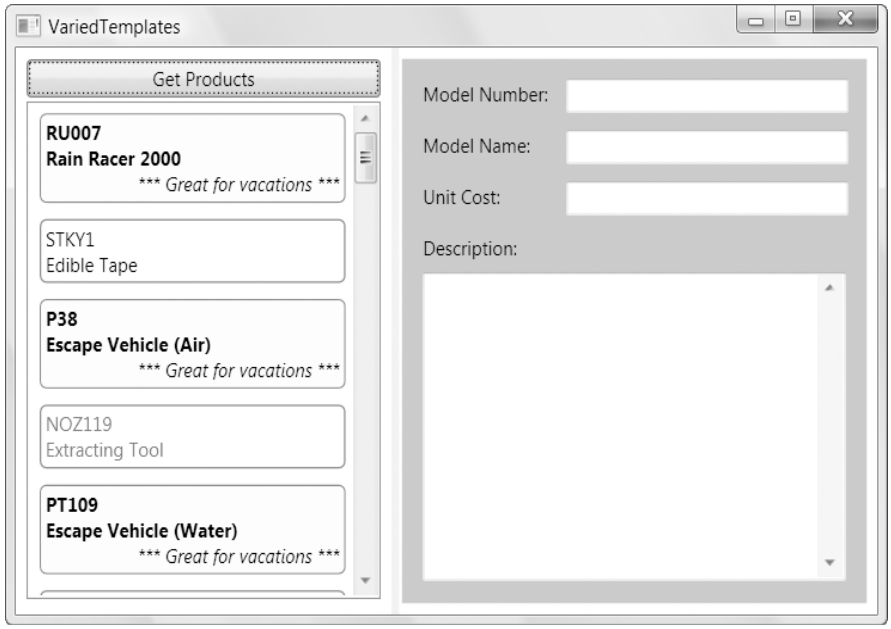

Figure 20-13. *A list with two data templates*

■ **Tip** One disadvantage with this approach is that you'll probably be forced to create multiple templates that are similar. If your templates are complex, this can create a lot of duplication. For best maintainability, you shouldn't create more than a few templates for a single list—instead, use triggers and styles to apply different formatting to your templates.

Templates and Selection

There's a small but irritating quirk in the previous template example. The problem is that the templates you've seen don't take selection into account.

If you select an item in the list, WPF automatically sets the Foreground and Background properties of the item container (in this case, the ListBoxItem object). The foreground is white, and the background is blue. The Foreground property uses property inheritance, so any elements you've added to your template automatically acquire the new white color, unless you've explicitly specified a new color. The Background color doesn't use property inheritance, but the default Background value is Transparent. If you have a transparent border, for example, the new blue background shows through. Otherwise, the color you've set in the template still applies.

This mishmash can alter your formatting in a way you might not intend. Figure 20-14 shows an example.

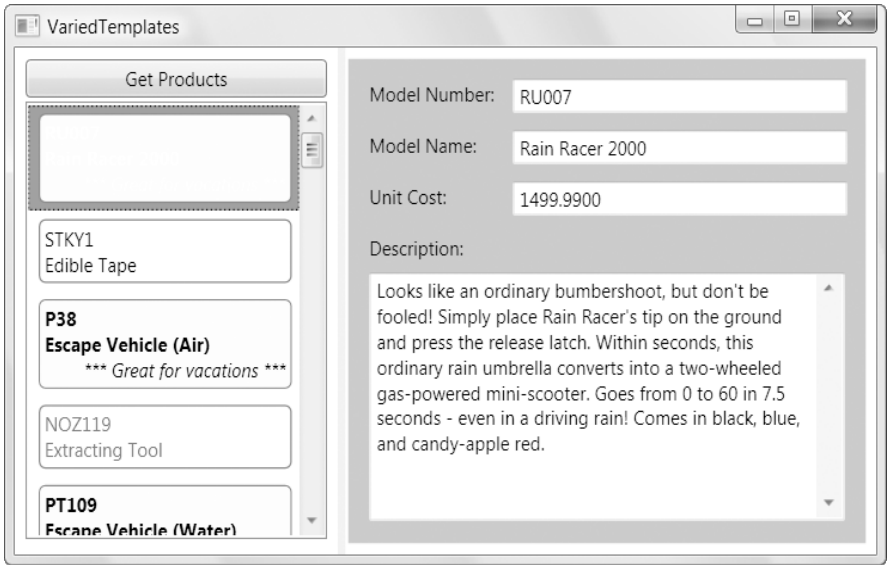

Figure 20-14. *Unreadable text in a highlighted item*

You could hard-code all your colors to avoid this problem, but then you'll face another challenge. The only indication that an item is selected will be the blue background around your curved border.

To solve this problem, you need to use the familiar ItemContainerStyle property to apply different formatting to the selected item:

```
<ListBox Name="lstProducts" HorizontalContentAlignment="Stretch">
  <ListBox.ItemContainerStyle>
    <Style>
      <Setter Property="Control.Padding" Value="0"></Setter>
      <Style.Triggers>
        <Trigger Property="ListBoxItem.IsSelected" Value="True">
          <Setter Property="ListBoxItem.Background" Value="DarkRed" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </ListBox.ItemContainerStyle>
</ListBox>
```

This trigger applies a dark red background to the selected item. Unfortunately, this code doesn't have the desired effect for a list that uses templates. That's because these templates include elements with a different background color that's displayed over the dark red background. Unless you make everything transparent (and allow the red color to wash through your entire template), you're left with a thin red edge around the margin area of your template.

The solution is to explicitly bind the background in part of your template to the value of the ListBoxItem.Background property. This makes sense—after all, you've now gone to the work of choosing the right background color to highlight the selected item. You just need to make sure it appears in the right place.

The markup you need to implement this solution is a bit messy. That's because you can't make do with an ordinary binding expression, which can simply bind to a property in the current data object (in this case, the Product object). Instead, you need to grab the background from the item container (in this case, the ListBoxItem). This involves using the Binding.RelativeSource property to search up the element tree for the first matching ListBoxItem object. Once that element is found, you can grab its background color and use it accordingly.

Here's the finished template, which uses the selected background in the curved border region. The Border element is placed inside a Grid with a white background, which ensures that the selected color does not appear in the margin area outside the curved border. The result is the much slicker selection style shown in Figure 20-15.

```
<DataTemplate>
  <Grid Margin="0" Background="White">
    <Border Margin="5" BorderThickness="1"
     BorderBrush="SteelBlue" CornerRadius="4"
     Background="{Binding Path=Background, RelativeSource={
                            RelativeSource
                            Mode=FindAncestor,
                            AncestorType={x:Type ListBoxItem}
                          }}" >
      <Grid Margin="3">
        <Grid.RowDefinitions>
          <RowDefinition></RowDefinition>
          <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <TextBlock FontWeight="Bold" Text="{Binding Path=ModelNumber}"></TextBlock>
        <TextBlock Grid.Row="1" Text="{Binding Path=ModelName}"></TextBlock>
      </Grid>
    </Border>
  </Grid>
</DataTemplate>
```

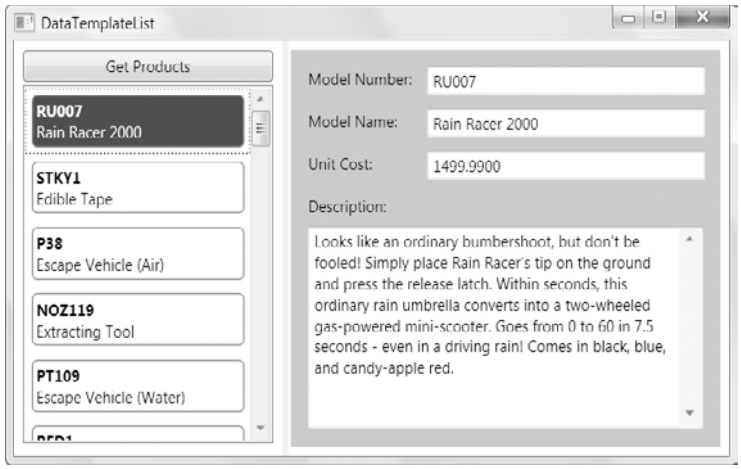

Figure 20-15. *Highlighting a selected item*

Selection and SnapsToDevicePixels

You should make one other change to ensure your template displays perfectly on computers with different system DPI settings (such as 120 dpi rather than the standard 96 dpi). You should set the ListBox.SnapsToDevicePixels property to true. This ensures that the edge of the list doesn't use anti-aliasing if it falls in between pixels.

If you don't set SnapsToDevicePixels to true, it's possible that you'll get a trace of the familiar blue border creeping in between the edge of your template and the edge of the containing ListBox control. (For more information about fractional pixels and why they occur when the system DPI is set to a value other than 96 dpi, see the discussion about WPF's device-independent measuring system in Chapter 1.)

This approach—using a binding expression to alter a template—works well if you can pull the property value you need out of the item container. For example, it's a great technique if you want to get the background and foreground color of a selected item. However, it isn't as useful if you need to alter the template in a more profound way.

For example, consider the list of product shown in Figure 20-16. When you select a product from this list, that item is expanded from a single-line text display to a box with a picture and full description. This example also combines several of the techniques you've already seen, including showing image content in a template and using data binding to set the background color of the Border element when an item is selected.

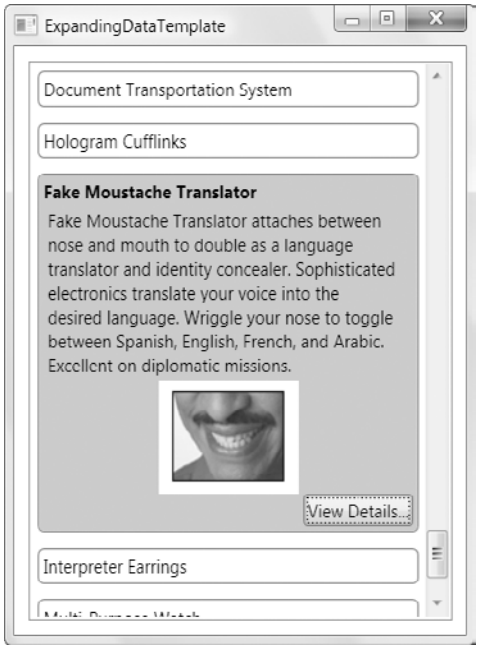

Figure 20-16. *Expanding a selected item*

To create this sort of list, you need to use a variation of the technique used in the previous example. You still need to use the RelativeSource property of a Binding to search for the current ListBoxItem. However, now you don't want to pull out its background color. Instead, you want to examine whether it's selected. If it isn't, you can hide the extra information by setting its Visibility property.

This technique is similar to the previous example but not exactly the same. In the previous example, you were able to bind directly to the value you wanted so that the background of the ListBoxItem became the background of the Border object. But in this case, you need to consider the ListBoxItem.IsSelected property and set the Visibility property of another element. The data types don't match—IsSelected is a Boolean value, while Visibility takes a value from the Visibility enumeration. As a result, you can't bind the Visibility property to the IsSelected property (at least, not without the help of a custom value converter). The solution is to use a data trigger so that when the IsSelected property is changed in the ListBoxItem, you modify the Visibility property of your container.

The place in your markup where you put the trigger is also different. It's no longer convenient to place the trigger in the ItemContainerStyle, because you don't want to change the visibility of the entire item. Instead, you want to hide just a single section, so the trigger needs to be part of a style that applies to just one container.

Here's a slightly simplified version of the template that doesn't have the automatically expanding behavior yet. Instead, it shows all the information (including the picture and description) for every product in the list.

```
<DataTemplate>
  <Border Margin="5" BorderThickness="1" BorderBrush="SteelBlue"
   CornerRadius="4">
    <StackPanel Margin="3">
      <TextBlock Text="{Binding Path=ModelName}"></TextBlock>
      <StackPanel>
        <TextBlock Margin="3" Text="{Binding Path=Description}"
         TextWrapping="Wrap" MaxWidth="250" HorizontalAlignment="Left"></TextBlock>
        <Image Source=
"{Binding Path=ProductImagePath, Converter={StaticResource ImagePathConverter}}">
        </Image>
        <Button FontWeight="Regular" HorizontalAlignment="Right" Padding="1"
         Tag="{Binding}">View Details...</Button>
      </StackPanel>
    </StackPanel>
  </Border>
</DataTemplate>
```

Inside the Border is a StackPanel that holds all the content. Inside that StackPanel is a second StackPanel that holds the content that should be shown only for selected items, which includes the description, image, and button. To hide this information, you need to set the style of the inner StackPanel using a trigger, as shown here:

```
<StackPanel>
  <StackPanel.Style>
    <Style>
      <Style.Triggers>
        <DataTrigger
          Binding="{Binding Path=IsSelected, RelativeSource={
                            RelativeSource
                            Mode=FindAncestor,
                            AncestorType={x:Type ListBoxItem}
                          }}"
          Value="False">
```

```
          <Setter Property="StackPanel.Visibility" Value="Collapsed" />
        </DataTrigger>
      </Style.Triggers>
    </Style>
  </StackPanel.Style>

  <TextBlock Margin="3" Text="{Binding Path=Description}"
   TextWrapping="Wrap" MaxWidth="250" HorizontalAlignment="Left"></TextBlock>
  <Image Source=
"{Binding Path=ProductImagePath, Converter={StaticResource ImagePathConverter}}">
  </Image>
  <Button FontWeight="Regular" HorizontalAlignment="Right" Padding="1"
   Tag="{Binding}">View Details...</Button>
</StackPanel>
```

In this example, you need to use a DataTrigger instead of an ordinary trigger, because the property you need to evaluate is in an ancestor element (the ListBoxItem), and the only way to access it is using a data binding expression.

Now, when the ListBoxItem.IsSelected property changes to False, the StackPanel.Visibility property is changed to Collapsed, hiding the extra details.

■ **Note** Technically, the expanded details are always present, just hidden. As a result, you'll experience the extra overhead of generating these elements when the list is first created, not when an item is selected. This doesn't make much difference in the current example, but this design could have a performance effect if you use it for an extremely long list with a complex template.

Changing Item Layout

Data templates give you remarkable control over every aspect of item presentation. However, they don't allow you to change how the items are organized with respect to each other. No matter what templates and styles you use, the ListBox puts each item into a separate horizontal row and stacks each row to create the list.

You can change this layout by replacing the container that the list uses to lay out its children. To do so, you set the ItemsPanelTemplate property with a block of XAML that defines the panel you want to use. This panel can be any class that derives from System.Windows.Controls.Panel.

The following uses a WrapPanel to wrap items across the available width of the ListBox control (as shown in Figure 20-17):

```
<ListBox Margin="7,3,7,10" Name="lstProducts"
 ItemTemplate="{StaticResource ItemTemplate}"
 ScrollViewer.HorizontalScrollBarVisibility="Disabled">
  <ListBox.ItemsPanel>
    <ItemsPanelTemplate>
      <WrapPanel></WrapPanel>
    </ItemsPanelTemplate>
  </ListBox.ItemsPanel>
</ListBox>
```

For this approach to work, you must also set the attached ScrollViewer.HorizontalScrollBarVisibility property to Disabled. This ensures that the ScrollViewer (which the ListBox uses automatically) never uses a horizontal scroll bar. Without this detail, the WrapPanel will be given infinite width in which to lay out its items, and this example becomes equivalent to a horizontal StackPanel.

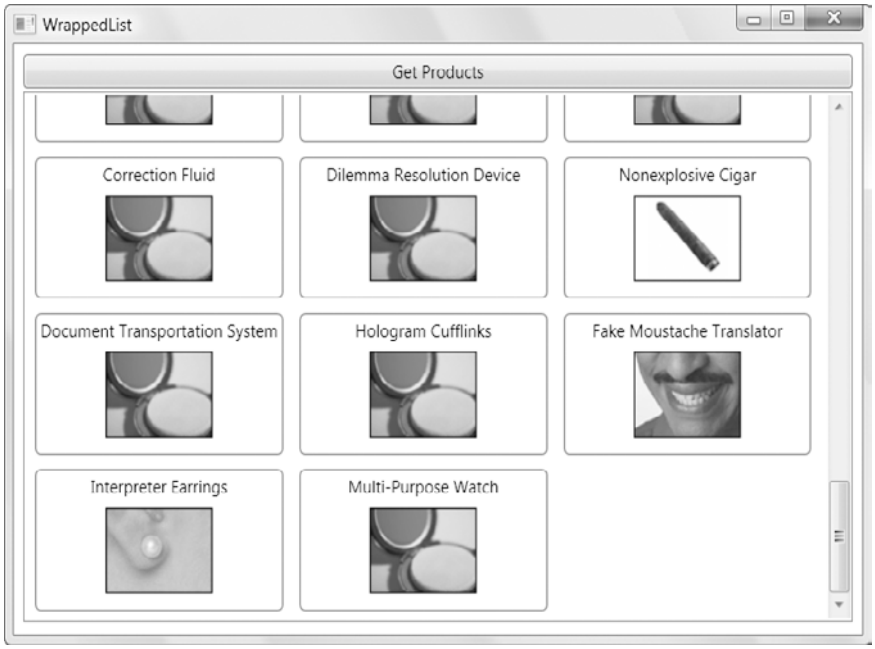

Figure 20-17. *Tiling items in the display area of a list*

There's one caveat with this approach. Ordinarily, most list controls use the VirtualizingStackPanel rather than the standard StackPanel. As discussed in Chapter 19, the VirtualizingStackPanel ensures that large lists of bound data are handled efficiently. When you use the VirtualizingStackPanel, it creates the elements that are required to show the set of currently visible items. When you use the StackPanel, it creates the elements that are required for the entire list. If your data source includes thousands of items (or more), the VirtualizingStackPanel will use far less memory. It will also perform better when you are filling the list and when the user is scrolling through it, because there's far less work for WPF's layout system to do.

Thus, you shouldn't replace set a new ItemsPanelTemplate unless you're using your list to show a fairly modest amount of data. If you're on the borderline—for example, you're showing only a couple hundred items but you have an extremely complex template—you can profile both approaches, see how performance and memory usage changes, and decide which strategy is best.

Incidentally, VirtualizingStackPanel inherits from the abstract class VirtualizingPanel. If you want to use a different type of panel without sacrificing virtualization support, you can derive your own custom panel class from VirtualizingPanel. Unfortunately, creating a reliable, professional-level virtualizing panel is difficult and beyond the scope of this book. If you'd like the challenge, you can get started by reading a high-level article at `http://tinyurl.com/mqtrdu`. Or, you can purchase one of the many third-party virtualizing panels that component vendors provide for WPF.

The ComboBox

Although styles and data templates are built into the ItemsControl class and supported by all the WPF list controls, so far all the examples you've seen have used the standard ListBox. There's nothing wrong with this fact—after all, the ListBox is thoroughly customizable and can easily handle lists of check boxes, images, formatted text, or a combination of all these types of content. However, other lists controls do introduce some new features. In Chapter 22, you'll learn about the frills of the ListView, TreeView, and DataGrid. But even the lowly ComboBox has a few extra considerations, and those are the details you'll explore in this section of this chapter.

Like the ListBox, the ComboBox is a descendant of the Selector class. Unlike the ListBox, the ComboBox is built out of two pieces: a selection box that shows the currently selected item and a drop-down list where you can choose that item. The drop-down list appears when you click the drop-down arrow at the edge of the combo box. Or, if your combo box is in read-only mode (the default), you can open the drop-down list by clicking anywhere in the selection box. Finally, you can programmatically open or close the drop-down list by setting the IsDropDownOpen property.

Ordinarily, the ComboBox control shows a read-only combo box, which means you can use it to select an item but can type in arbitrary text of your own. However, you can change this behavior by setting the IsReadOnly property to false and the IsEditable property to true. Now, the selection box becomes a text box, and you can type in whatever text you want.

The ComboBox control provides a rudimentary form of autocomplete that completes entries as you type. (This shouldn't be confused with the fancier autocomplete that you see in programs such as Internet Explorer, which shows a whole *list* of possibilities under the current text box.) Here's how it works—as you type in the ComboBox control, WPF fills in the remainder of the selection box with the first matching autocomplete suggestion. For example, if you type **Gr** and your list contains *Green*, the combo box will fill in the letters *een*. The autocomplete text is selected, so you'll automatically overwrite it if you keep typing.

If you don't want the autocomplete behavior, simply set the ComboBox.IsTextSearchEnabled property to false. This property is inherited from the base ItemsControl class, and it applies to many other list controls. For example, if IsTextSearchEnabled is set to true in a ListBox, you can type the first level of an item to jump to that position.

■ **Note** WPF doesn't include any features for using the system-tracked autocomplete lists, such as the list of recent URLs and files. It also doesn't provide support for drop-down autocomplete lists.

So far, the behavior of the ComboBox is quite straightforward. However, it changes a bit if your list contains more complex objects rather than simple strings of text.

You can place more complex objects in a ComboBox in two ways. The first option is to add them manually. As with the ListBox, you can place any content you want in a ComboBox. For example, if you want a list of images and text, you'd simply place the appropriate elements in a StackPanel and wrap that StackPanel in a ComboBoxItem object. More practically, you can use a data template to insert the content from a data object into a predefined group of elements.

When using nontext content, it's not as obvious what the selection box should contain. If the IsEditable property is false (the default), the selection box will show an exact visual copy of the item. For example, Figure 20-18 shows a ComboBox that uses a data template that incorporates text and image content.

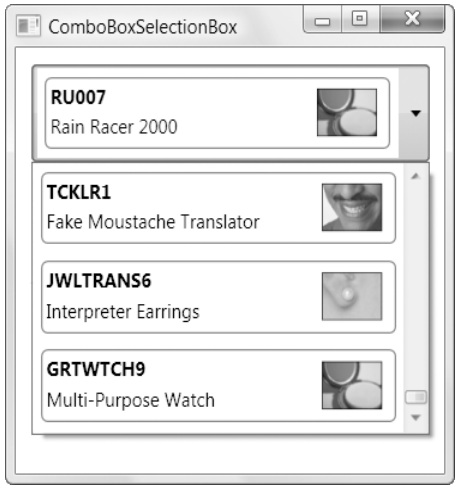

Figure 20-18. *A read-only ComboBox that uses templates*

■ **Note** The important detail is what the combo box is displaying as its content, not what it has as its data source. For example, imagine you fill a ComboBox control with Product objects and set the DisplayMemberPath property to ModelName so the combo box shows the ModelName property of each item. Even though the combo box retrieves its information from a group of Product objects, your markup creates an ordinary text list. As a result, the selection box will behave the way you expect it to behave. It will show the ModelName of the current product, and if IsEditable is true and IsReadOnly is false, it will allow you to edit that value.

The user won't be able to interact with the content that appears in the selection box. For example, if the content of the currently selected item includes a text box, you won't be able to type in it. If the currently selected item includes a button, you won't be able to click it. Instead, clicking the selection box will simply open the drop-down list. (Of course, there are countless good usability reasons not to put user-interactive controls in a drop-down list in the first place.)

If the IsEditable property is true, the behavior of the ComboBox control changes. Instead of showing a copy of the selected item, the selection box displays a textual representation of it. To create this textual representation, WPF simply calls ToString() on the item. Figure 20-19 shows an example with the same combo box that's shown in Figure 20-19. In this case, the display text DataBinding.Product is simply the fully qualified class name of the currently selected Product object, which is the default ToString() implementation unless you override it in your data class.

Figure 20-19. *An editable ComboBox that uses templates*

The easiest option to correct this problem is to set the attached TextSearch.TextPath property to indicate the property that should be used for the content of the selection box. Here's an example:

```
<ComboBox IsEditable="True" IsReadOnly="True" TextSearch.TextPath="ModelName" ...>
```

Although IsEditable must be true, it's up to you whether you set IsReadOnly to false (to allow editing of that property) or true (to prevent the user from typing in arbitrary text). Figure 20-20 shows the result.

■ **Tip** What if you want to show richer content than a simple piece of text but you still want the content in the selection box to be different from the content in the drop-down list? The ComboBox includes a SelectionBoxItemTemplate property that defines the template that's used for the selection box. Unfortunately, the SelectionBoxItemTemplate is read-only. It's automatically set to match the current item, and you can't supply a different template. However, you could create an entirely new ComboBox control template that doesn't use the SelectionBoxItemTemplate at all. Instead, this control template could hard-code the selection box template or could retrieve it from the Resources collection in the window.

Figure 20-20. *Displaying a property in the selection box*

The Last Word

In this chapter, you delved deeper into data binding, one of the key pillars of WPF.

In the past, many of the scenarios you considered in this chapter would be handled using code. In WPF, the data binding model (in conjunction with value converters, styles, and data templates) allows you to do much more work declaratively. In fact, data binding is nothing less than an all-purpose way to display any type of information, regardless of where it's stored, how you want to displayed, or whether it's editable. Sometimes, this data will be drawn from a back-end database. In other cases, it may come from a web service, a remote object, or the file system, or it may be generated entirely in code. Ultimately, it won't matter—as long as the data model remains constant, your user interface code and binding expressions will remain the same.

CHAPTER 21

Data Views

Now that you've explored the art of converting data, applying styles to the items in a list, and building data templates, you're ready to move on to *data views*, which work behind the scenes to coordinate collections of bound data. Using data views, you can add navigation logic and implement filtering, sorting, and grouping.

The View Object

When you bind a collection (or a DataTable) to an ItemsControl, a data view is quietly created behind the scenes. This view sits between your data source and the bound control. The data view is a window into your data source. It tracks the current item, and it supports features such as sorting, filtering, and grouping. These features are independent of the data object itself, which means you can bind the same data in different ways in different portions of a window (or different parts of your application). For example, you could bind the same collection of products to two different lists but filter them to show different records.

The view object that's used depends on the type of data object. All views derive from CollectionView, but two specialized implementations derive from CollectionView: ListCollectionView and BindingListCollectionView. Here's how it works:

- If your data source implements IBindingList, a BindingListCollectionView is created. This happens when you bind an ADO.NET DataTable.
- If your data source doesn't implement IBindingList but it implements IList, a ListCollectionView is created. This happens when you bind an ObservableCollection, like the list of products.
- If your data source doesn't implement IBindingList or IList but it implements IEnumerable, you get a basic CollectionView.

Tip Ideally, you'll avoid the third scenario. The CollectionView offers poor performance for large items and operations that modify the data source (such as insertions and deletions). As you learned in Chapter 19, if you're not binding to an ADO.NET data object, it's almost always easiest to use the ObservableCollection class.

Retrieving a View Object

To get ahold of a view object that's currently in use, you use the static GetDefaultView() method of the System.Windows.Data.CollectionViewSource class. When you call GetDefaultView(), you pass in the data source—the collection or DataTable that you're using. Here's an example that gets the view for the collection of products that's bound to the list:

```
ICollectionView view = CollectionViewSource.GetDefaultView(lstProducts.ItemsSource);
```

The GetDefaultView() method always returns an ICollectionView reference. It's up to you to cast the view object to the appropriate class, such as a ListCollectionView or BindingListCollectionView, depending on the data source.

```
ListCollectionView view =
  (ListCollectionView)CollectionViewSource.GetDefaultView(lstProducts.ItemsSource);
```

Navigating with a View

One of the simplest things you can do with a view object is determine the number of items in the list (through the Count property) and get a reference to the current data object (CurrentItem) or current position index (CurrentPosition). You can also use a handful of methods to move from one record to another, such as MoveCurrentToFirst(), MoveCurrentToLast(), MoveCurrentToNext(), MoveCurrentToPrevious(), and MoveCurrentToPosition(). So far, you haven't needed these details because all the examples you've seen have used the list to allow the user to move from one record to the next. But if you want to create a record browser application, you might want to supply your own navigation buttons. Figure 21-1 shows one example.

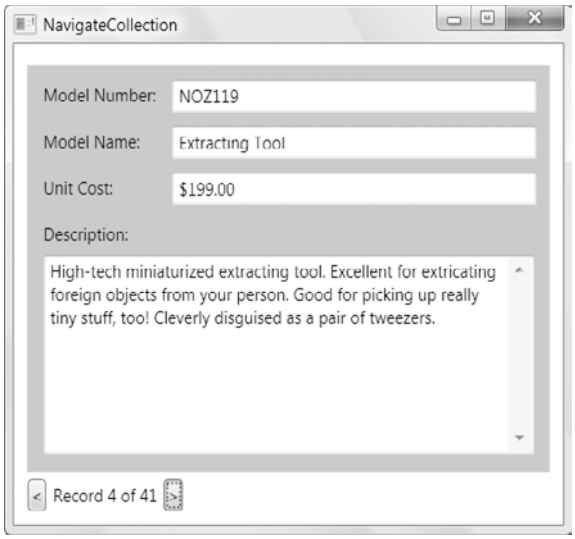

Figure 21-1. *A record browser*

The bound text boxes that show the data for the bound product stay the same. They need only to indicate the appropriate property, as shown here:

```
<TextBlock Margin="7">Model Number:</TextBlock>
<TextBox Margin="5" Grid.Column="1" Text="{Binding Path=ModelNumber}"></TextBox>
```

However, this example doesn't include any list control, so it's up to you to take control of the navigation. To simplify life, you can store a reference to the view as a member variable in your window class:

```
private ListCollectionView view;
```

In this case, the code casts the view to the appropriate view type (ListCollectionView) rather than using the ICollectionView interface. The ICollectionView interface provides most of the same functionality, but it lacks the Count property that gives the total number of items in the collection.

When the window first loads, you can get the data, place it in the DataContext of the window, and store a reference to the view:

```
ICollection<Products> products = App.StoreDB.GetProducts();
this.DataContext = products;

view = (ListCollectionView)CollectionViewSource.GetDefaultView(this.DataContext);
view.CurrentChanged += new EventHandler(view_CurrentChanged);
```

The second line does all the magic needed to show your collection of items in the window. It places the whole collection of Product objects in the DataContext. The bound controls on the form will search up the element tree until they find this object. Of course, you want the binding expressions to bind to the current item in the collection, not the collection itself, but WPF is smart enough to figure this out automatically. It automatically supplies them with the current item, so you don't need a stitch of extra code.

The previous example has one additional code statement. It connects an event handler to the CurrentChanged event of the view. When this event fires, you can perform a few useful actions, such as enabling or disabling the previous and next buttons depending on the current position and displaying the current position in a TextBlock at the bottom of the window.

```
private void view_CurrentChanged(object sender, EventArgs e)
{
    lblPosition.Text = "Record " + (view.CurrentPosition + 1).ToString() +
      " of " + view.Count.ToString();
    cmdPrev.IsEnabled = view.CurrentPosition > 0;
    cmdNext.IsEnabled = view.CurrentPosition < view.Count - 1;
}
```

This code seems like a candidate for data binding and triggers. However, the logic is just a bit too complex (partly because you need to add 1 to the index to get the record position number that you want to display).

The final step is to write the logic for the previous and next buttons. Because these buttons are automatically disabled when they don't apply, you don't need to worry about moving before the first item or after the last item.

```
private void cmdNext_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToNext();
}

private void cmdPrev_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToPrevious();
}
```

For an interesting frill, you can add a list control to this form so the user has the option of stepping through the records one at a time with the buttons or using the list to jump directly to a specific item (as shown in Figure 21-2).

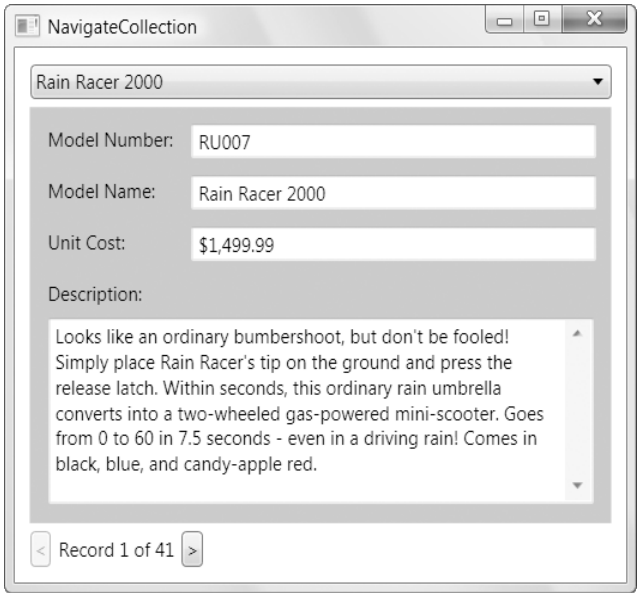

Figure 21-2. *A record browser with a drop-down list*

In this case, you need a ComboBox that uses the ItemsSource property (to get the full list of products) and uses a binding on the Text property (to show the right item):

```
<ComboBox Name="lstProducts" DisplayMemberPath="ModelName"
 Text="{Binding Path=ModelName}"
 SelectionChanged="lstProducts_SelectionChanged"></ComboBox>
```

When you first retrieve the collection of products, you'll bind the list:

```
lstProducts.ItemsSource = products;
```

This might not have the effect you expect. By default, the selected item in an ItemsControl is not synchronized with the current item in the view. That means that when you make a new selection from the list, you aren't directed to the new record—instead, you end up modifying the ModelName property of the current record. Fortunately, there are two easy approaches to solve the problem.

The brute-force approach is to simply move to the new record whenever an item is selected in the list. Here's the code that does it:

```
private void lstProducts_SelectionChanged(object sender, RoutedEventArgs e)
{
    view.MoveCurrentTo(lstProducts.SelectedItem);
}
```

A simpler solution is to set the ItemsControl.IsSynchronizedWithCurrentItem to true. That way, the currently selected item is automatically synchronized to match the current position of the view with no code required.

Using a Lookup List for Editing

The ComboBox provides a handy way to edit record values. In the current example, it doesn't make much sense—after all, there's no reason to give one product the same name as another product. However, it's not difficult to think of other scenarios where the ComboBox is a great editing tool.

For example, you might have a field in your database that accepts one of a small set of preset values. In this case, use a ComboBox, and bind it to the appropriate field using a binding expression for the Text property. However, fill the ComboBox with the allowable values by setting its ItemsSource property to point to the list you've defined. And if you want to display the values in the list one way (say, as text) but store them another way (as numeric codes), just add a value converter to your Text property binding.

Another case where a lookup list makes sense is when dealing with related tables. For example, you might want to allow the user to pick the category for a product using a list of all the defined categories. The basic approach is the same: set the Text property to bind to the appropriate field, and fill in the list of options with the ItemsSource property. If you need to convert low-level unique IDs into more meaningful names, use a value converter.

Creating a View Declaratively

The previous example used a simple pattern that you'll see throughout this chapter. The code retrieves the view you want to use and then modifies it programmatically. However, you have another choice—you can construct a CollectionViewSource declaratively in XAML markup and then bind the CollectionViewSource to your controls (such as the list).