CHAPTER 20

Formatting Bound Data

In Chapter 19, you learned the essentials of WPF data binding—how to pull information out of an object and display it in a window, with little code required. Along the way, you considered how to make that information editable, how to deal with collections of data objects, and how to perform validation to catch bad edits. However, there's still a lot more to learn.

In this chapter, you'll continue your exploration by tackling several subjects that will allow you to build better bound windows. First you'll look at data conversion, the powerful and extensible system WPF uses to examine values and convert them. As you'll discover, this process of conversion goes far beyond simple conversion, giving you the ability to apply conditional formatting and deal with images, files, and other types of specialized content.

Next you'll look at how you can format entire lists of data. You'll review the infrastructure that supports bound lists, starting with the base ItemsControl class. Then you'll learn how to refine the appearance of lists with styles, along with triggers that apply alternating formatting and selection highlighting. Finally, you'll use the most powerful formatting tool of all, *data templates*, which let you customize the way each item is shown in an ItemsControl. Data templates are the secret to converting a basic list into a rich data presentation tool complete with custom formatting, picture content, and additional WPF controls.

Data Binding Redux

In most data-binding scenarios, you aren't binding to a single object but to an entire collection. Figure 20-1 shows a familiar example—a form with a list of products. When the user selects a product, its details appear on the right.

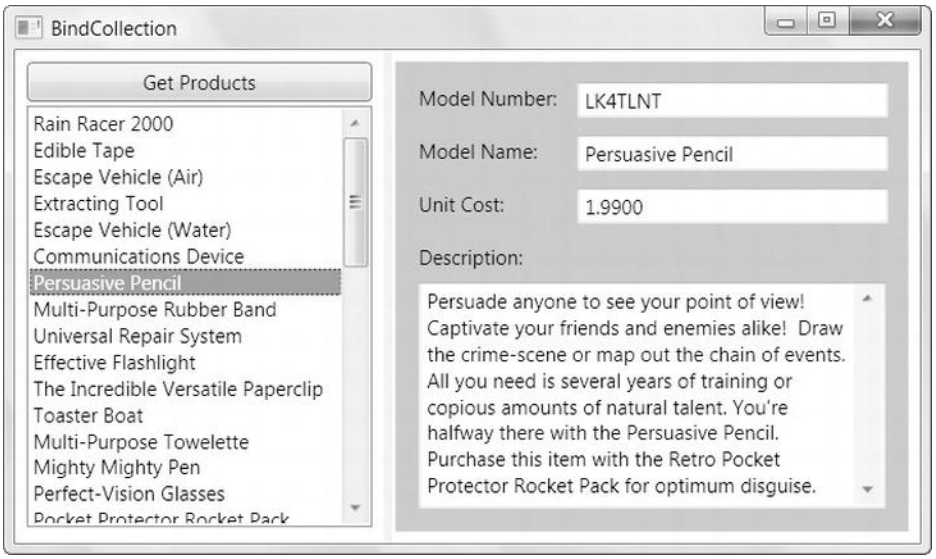

Figure 20-1. *Browsing a collection of products*

In Chapter 19, you learned to build exactly this sort of form. Here's a quick review of the basic steps:

1. First you need to create the list of items, which you can show in an ItemsControl. Set the DisplayMemberPath to indicate the property (or field) you want to show for each item in the list. This list shows the model name of each item:

   ```
   <ListBox Name="lstProducts" DisplayMemberPath="ModelName"></ListBox>
   ```

2. To fill the list with data, set the ItemsSource property to your collection (or DataTable). Typically, you'll perform this step in code when your window loads or the user clicks a button. In this example, the ItemsControl is bound to an ObservableCollection of Product objects.

   ```
   ObservableCollection<Product> products = App.StoreDB.GetProducts();
   lstProducts.ItemsSource = products;
   ```

3. To show item-specific information, add as many elements as you need, each with a binding expression that identifies the property or field you want to display. In this example, each item in the collection is a Product object. Here's an example that shows the model number of an item by binding to the Product.ModelNumber property:

   ```
   <TextBox Text="{Binding Path=ModelNumber}"></TextBox>
   ```

4. The easiest way to connect the item-specific elements to the currently selected item is to wrap them in a single container. Set the DataContext property of the container to refer to the selected item in the list:

   ```
   <Grid DataContext="{Binding ElementName=lstProducts, Path=SelectedItem}">
   ```

So far, this is all review. However, what you haven't yet considered is how to tailor the appearance of the data list and the data fields. For example, you don't yet know how to format numeric values, how to create a list that shows multiple pieces of information at once (and arranges these pieces in a pleasing way), and how to deal with nontext content, such as picture data. In this chapter, you'll cover all these tasks as you begin to build more-respectable data forms.

Data Conversion

In a basic binding, the information travels from the source to the target without any change. This seems logical, but it's not always the behavior you want. Often your data source might use a low-level representation that you don't want to display directly in your user interface. For example, you might have numeric codes you want to replace with human-readable strings, numbers that need to be cut down to size, dates that need to be displayed in a long format, and so on. If so, you need a way to convert these values into the right display form. And if you're using a two-way binding, you also need to do the converse—take user-supplied data and convert it to a representation suitable for storage in the appropriate data object.

Fortunately, WPF has two tools that can help you:

String formatting: This feature allows you to convert data that's represented as text—for example, strings that contain dates and numbers—by setting the Binding.StringFormat property. It's a convenient technique that works for at least half of all formatting tasks.

Value converters: This is a far more powerful (and somewhat more complicated) feature that lets you convert any type of source data into any type of object representation, which you can then pass on to the linked control.

In the following sections, you'll consider both approaches.

Using the StringFormat Property

String formatting is the perfect tool for formatting numbers that need to be displayed as text. For example, consider the UnitCost property from the Product class introduced in the previous chapter. UnitCost is stored as a decimal, and, as a result, when it's displayed in a text box, you'll see values such as 3.9900. Not only does this display format show more decimal places than you'd probably like, it also leaves out the currency symbol. A more intuitive representation would be the currency-formatted value $3.99.

The easiest solution is to set the Binding.StringFormat property. WPF will use the format string to convert the raw text to its display value, just before it appears in the control. Just as important, WPF will (in most cases) use this string to perform the reverse conversion, taking any edited data and using it to update the bound property.

When setting the Binding.StringFormat property, you use standard .NET format strings, in the form {0:C}. Here, the *0* represents the first value, and the *C* refers to the format string you want to apply—which is, in this case, the standard local-specific currency format, which translates 3.99 to $3.99 on a US computer. The entire expression is wrapped in curly braces.

Here's an example that applies the format string to the UnitCost field so that it's displayed as a currency value:

```
<TextBox Margin="5" Grid.Row="2" Grid.Column="1"
 Text="{Binding Path=UnitCost, StringFormat={}{0:C}}">
</TextBox>
```

You'll notice that the StringFormat value is preceded with the curly braces {}. In full, it's {}{0:C} rather than just {0:C}. The slightly unwieldy pair of braces at the beginning is required to escape the string. Otherwise, the XAML parser can be confused by the curly brace at the beginning of {0:C}.

Incidentally, the {} escape sequence is required only when the StringFormat value begins with a brace. Consider this example, which adds a literal sequence of text before each formatted value:

```
<TextBox Margin="5" Grid.Row="2" Grid.Column="1"
 Text="{Binding Path=UnitCost, StringFormat=The value is {0:C}.}">
</TextBox>
```

This expression converts a value such as 3.99 to "The value is $3.99." Because the first character in the StringFormat value is an ordinary letter, not a brace, the initial escape sequence isn't required. However, this format string works in one direction only. If the user tries to supply an edited value that includes this literal text (such as "The value is $4.25"), the update will fail. On the other hand, if the user performs an edit with the numeric characters only (4.25) or with the numeric characters and the currency symbol ($4.25), the edit will succeed, and the binding expression will convert it to the display text "The value is $4.25" and show that in the text box.

To get the results you want with the StringFormat property, you need the right format string. You can learn about all the format strings that are available in the Visual Studio help. However, Table 20-1 and Table 20-2 show some of the most common options you'll use for numeric and date values, respectively. And here's a binding expression that uses a custom date format string to format the OrderDate property:

```
<TextBlock Text="{Binding Date, StringFormat={}{0:MM/dd/yyyy}}"></TextBlock>
```

Table 20-1. *Format Strings for Numeric Data*

Type	Format String	Example
Currency	C	$1,234.50. Parentheses indicate negative values: ($1,234.50). The currency sign is locale-specific.
Scientific (Exponential)	E	1.234.50E+004.
Percentage	P	45.6%.
Fixed Decimal	F?	Depends on the number of decimal places you set. F3 formats values such as 123.400. F0 formats values such as 123.

Table 20-2. *Format Strings for Times and Dates*

Type	Format String	Format
Short Date	d	M/d/yyyy For example: 10/30/2008
Long Date	D	dddd, MMMM dd, yyyy For example: Wednesday, January 30, 2008
Long Date and Short Time	f	dddd, MMMM dd, yyyy HH:mm aa For example: Wednesday, January 30, 2008 10:00 AM
Long Date and Long Time	F	dddd, MMMM dd, yyyy HH:mm:ss aa For example: Wednesday, January 30, 2008 10:00:23 AM
ISO Sortable Standard	s	yyyy-MM-dd HH:mm:ss For example: 2008-01-30 10:00:23

Month and Day	M	MMMM dd For example: January 30
General	G	M/d/yyyy HH:mm:ss aa (depends on locale-specific settings) For example: 10/30/2008 10:00:23 AM

The WPF list controls also support string formatting for list items. To use it, you simply set the ItemStringFormat property of the list (which is inherited from the base ItemsControl class). Here's an example with a list of product prices:

```
<ListBox Name="lstProducts" DisplayMemberPath="UnitCost" ItemStringFormat="{0:C}">
</ListBox>
```

The formatting string is automatically passed down to the binding that grabs the text for each item.

Introducing Value Converters

The Binding.StringFormat property is created for simple, standard formatting with numbers and dates. But many data-binding scenarios need a more powerful tool, called a *value converter* class.

A value converter plays a straightforward role. It's responsible for converting the source data just before it's displayed in the target and (in the case of a two-way binding) converting the new target value just before it's applied back to the source.

■ **Note** This approach to conversion is similar to the way data binding worked in the world of Windows Forms with the Format and Parse binding events. The difference is that in a Windows Forms application, you could code this logic anywhere—you simply needed to attach both events to the binding. In WPF, this logic must be encapsulated in a value converter class, which makes for easier reuse.

Value converters are an extremely useful piece of the WPF data-binding puzzle. They can be used in several useful ways:

To format data to a string representation: For example, you can convert a number to a currency string. This is the most obvious use of value converters, but it's certainly not the only one.

To create a specific type of WPF object: For example, you could read a block of binary data and create a BitmapImage object that can be bound to an Image element.

To conditionally alter a property in an element based on the bound data: For example, you might create a value converter that changes the background color of an element to highlight values in a specific range.

Formatting Strings with a Value Converter

To get a basic idea of how a value converter works, it's worth revisiting the currency-formatting example you looked at in the previous section. Although that example used the Binding.StringFormat property, you can accomplish the same thing—and more—with a value converter. For example, you could round or

truncate values (changing 3.99 to 4), use number names (changing 1,000,000 into 1 million), or even add a dealer markup (multiplying 3.99 by 15%). You can also tailor the way that the reverse conversion works to change user-supplied values into the right data values in the bound object.

To create a value converter, you need to take four steps:

1. Create a class that implements IValueConverter.
2. Add the ValueConversion attribute to the class declaration, and specify the destination and target data types.
3. Implement a Convert() method that changes data from its original format to its display format.
4. Implement a ConvertBack() method that does the reverse and changes a value from display format to its native format.

Figure 20-2 shows how it works.

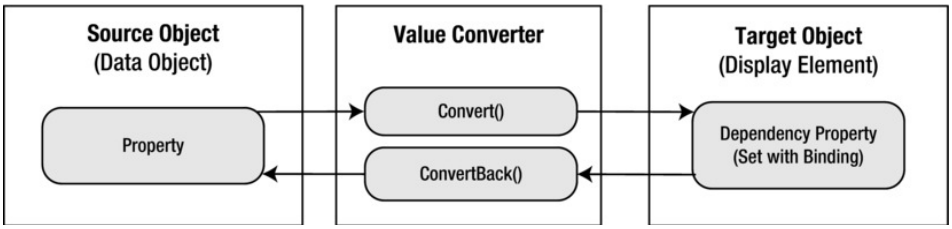

Figure 20-2. *Converting bound data*

In the case of the decimal-to-currency conversion, you can use the Decimal.ToString() method to get the formatted string representation you want. You simply need to specify the currency format string "C", as shown here:

```
string currencyText = decimalPrice.ToString("C");
```

This code uses the culture settings that apply to the current thread. A computer that's configured for the English (United States) region runs with a locale of en-US and displays currencies with the dollar sign ($). A computer that's configured for another locale might display a different currency symbol. (This is the same way that the {0:C} format string works when applied with the Binding.StringFormat property.) If this isn't the result you want (for example, you always want the dollar sign to appear), you can specify a culture by using the overload of the ToString() method shown here:

```
CultureInfo culture = new CultureInfo("en-US");
string currencyText = decimalPrice.ToString("C", culture);
```

Converting from the display format back to the number you want is a little trickier. The Parse() and TryParse() methods of the Decimal type are logical choices to do the work, but ordinarily they can't handle strings that include currency symbols. The solution is to use an overloaded version of the Parse() or TryParse() method that accepts a System.Globalization.NumberStyles value. If you supply NumberStyles.Any, you'll be able to successfully strip out the currency symbol, if it exists.

Here's the complete code for the value converter that deals with price values such as the Product.UnitCost property:

```
[ValueConversion(typeof(decimal), typeof(string))]
public class PriceConverter : IValueConverter
```

```
{
    public object Convert(object value, Type targetType, object parameter,
      CultureInfo culture)
    {
        decimal price = (decimal)value;
        return price.ToString("C", culture);
    }

    public object ConvertBack(object value, Type targetType, object parameter,
      CultureInfo culture)
    {
        string price = value.ToString(culture);

        decimal result;
        if (Decimal.TryParse(price, NumberStyles.Any, culture, out result))
        {
            return result;
        }
        return value;
    }
}
```

To put this converter into action, you need to begin by mapping your project namespace to an XML namespace prefix you can use in your markup. Here's an example that uses the namespace prefix *local* and assumes that your value converter is in the namespace DataBinding:

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

Typically, you'll add this attribute to the <Window> tag that holds all your markup.

Now you simply need to create an instance of the PriceConverter class and assign it to the Converter property of your binding. To do this, you need the more long-winded syntax shown here:

```
<TextBlock Margin="7" Grid.Row="2">Unit Cost:</TextBlock>
<TextBox Margin="5" Grid.Row="2" Grid.Column="1">
  <TextBox.Text>
    <Binding Path="UnitCost">
      <Binding.Converter>
        <local:PriceConverter></local:PriceConverter>
      </Binding.Converter>
    </Binding>
  </TextBox.Text>
</TextBox>
```

In many cases, the same converter is used for multiple bindings. In this case, it doesn't make sense to create an instance of the converter for each binding. Instead, create one converter object in the Resources collection, as shown here:

```
<Window.Resources>
  <local:PriceConverter x:Key="PriceConverter"></local:PriceConverter>
</Window.Resources>
```

Then you can point to it in your binding by using a StaticResource reference, as described in Chapter 10:

```
<TextBox Margin="5" Grid.Row="2" Grid.Column="1"
 Text="{Binding Path=UnitCost, Converter={StaticResource PriceConverter}}">
</TextBox>
```

Creating Objects with a Value Converter

Value converters are indispensable when you need to bridge the gap between the way data is stored in your classes and the way it's displayed in a window. For example, imagine you have picture data stored as a byte array in a field in a database. You could convert the binary data into a System.Windows.Media.Imaging.BitmapImage object and store that as part of your data object. However, this design might not be appropriate.

For example, you might need the flexibility to create more than one object representation of your image, possibly because your data library is used in both WPF applications and Windows Forms applications (which use the System.Drawing.Bitmap class instead). In this case, it makes sense to store the raw binary data in your data object and convert it to a WPF BitmapImage object by using a value converter. (To bind it to a form in a Windows Forms application, you'd use the Format and Parse events of the System.Windows.Forms.Binding class.)

■ **Tip** To convert a block of binary data into an image, you must first create a BitmapImage object and read the image data into a MemoryStream. Then you can call the BitmapImage.BeginInit() method, set its StreamSource property to point to your MemoryStream, and call EndInit() to finish loading the image.

The Products table from the Store database doesn't include binary picture data, but it does include a ProductImage field that stores the file name of an associated product image. In this case, there's even more reason to delay creating the image object. First, the image might not be available, depending on where the application's running. Second, there's no point in incurring the extra memory overhead for storing the image unless it's going to be displayed.

The ProductImage field includes the file name but not the full path of an image file, which gives you the flexibility to put the image files in any suitable location. The value converter has the task of creating a URI that points to the image file based on the ProductImage field and the directory you want to use. The directory is stored by using a custom property named ImageDirectory, which defaults to the current directory.

Here's the complete code for the ImagePathConverter that performs the conversion:

```
public class ImagePathConverter : IValueConverter
{
    private string imageDirectory = Directory.GetCurrentDirectory();
    public string ImageDirectory
    {
        get { return imageDirectory; }
        set { imageDirectory = value; }
    }

    public object Convert(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
    {
        string imagePath = Path.Combine(ImageDirectory,
          (string)value);
```

```
        return new BitmapImage(new Uri(imagePath));
    }

    public object ConvertBack(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
    {
        throw new NotSupportedException();
    }
}
```

To use this converter, begin by adding it to the Resources. In this example, the ImageDirectory property is not set, which means the ImagePathConverter defaults to the current application directory:

```
<Window.Resources>
  <local:ImagePathConverter x:Key="ImagePathConverter"></local:ImagePathConverter>
</Window.Resources>
```

Now it's easy to create a binding expression that uses this value converter:

```
<Image Margin="5" Grid.Row="2" Grid.Column="1" Stretch="None"
 HorizontalAlignment="Left" Source=
 "{Binding Path=ProductImagePath, Converter={StaticResource ImagePathConverter}}">
</Image>
```

This works because the Image.Source property expects an ImageSource object, and the BitmapImage class derives from ImageSource.

Figure 20-3 shows the result.

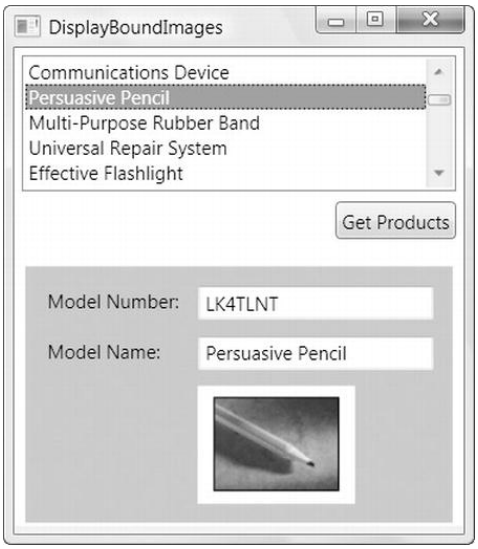

Figure 20-3. *Displaying bound images*

You might improve this example in a couple of ways. First, attempting to create a BitmapImage that points to a nonexistent file causes an exception, which you'll receive when setting the DataContext,

ItemsSource, or Source property. Alternatively, you can add properties to the ImagePathConverter class that allow you to configure this behavior. For example, you might introduce a Boolean SuppressExceptions property. If set to true, you could catch exceptions in the Convert() method and then return the Binding.DoNothing value (which tells WPF to temporarily act as though no data binding is set). Or you could add a DefaultImage property that takes a placeholder BitmapImage. The ImagePathConverter could then return the default image if an exception occurs.

You'll also notice that this converter supports only one-way conversion. That's because it's not possible to change the BitmapImage object and use that to update the image path. However, you could take an alternate approach. Rather than return a BitmapImage from the ImagePathConverter, you could simply return the fully qualified URI from the Convert() method, as shown here:

```
return new Uri(imagePath);
```

This works just as successfully, because the Image element uses a type converter to translate the Uri to the ImageSource object it really wants. If you take this approach, you could then allow the user to choose a new file path (perhaps using a TextBox that's set with the help of the OpenFileDialog class). You could then extract the file name in the ConvertBack() method and use that to update the image path that's stored in your data object.

Applying Conditional Formatting

Some of the most interesting value converters aren't designed to format data for presentation. Instead, they're intended to format some other appearance-related aspect of an element based on a data rule.

For example, imagine you want to flag high-priced items by giving them a different background color. You can easily encapsulate this logic with the following value converter:

```
public class PriceToBackgroundConverter : IValueConverter
{
    public decimal MinimumPriceToHighlight
    {
        get; set;
    }

    public Brush HighlightBrush
    {
        get; set;
    }

    public Brush DefaultBrush
    {
        get; set;
    }

    public object Convert(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
    {
        decimal price = (decimal)value;
        if (price >= MinimumPriceToHighlight)
            return HighlightBrush;
        else
            return DefaultBrush;
```

```
    }

    public object ConvertBack(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
    {
        throw new NotSupportedException();
    }
}
```

Once again, the value converter is carefully designed with reusability in mind. Rather than hard-coding the color highlights in the converter, they're specified in the XAML by the code that *uses* the converter:

```
<local:PriceToBackgroundConverter x:Key="PriceToBackgroundConverter"
  DefaultBrush="{x:Null}" HighlightBrush="Orange" MinimumPriceToHighlight="50">
</local:PriceToBackgroundConverter>
```

Brushes are used instead of colors so that you can create more-advanced highlight effects by using gradients and background images. And if you want to keep the standard, transparent background (so the background of the parent elements is used), just set the DefaultBrush or HighlightBrush property to null.

Now all that's left is to use this converter to set the background of some element, such as the Border that contains all the other elements:

```
<Border Background=
 "{Binding Path=UnitCost, Converter={StaticResource PriceToBackgroundConverter}}"
 ... >
```

OTHER WAYS TO APPLY CONDITIONAL FORMATTING

Using a custom value converter is only one of the ways to apply conditional formatting based on your data object. You can also use data triggers in a style, style selector, and template selector, all of which are described in this chapter. Each one of these approaches has its own advantages and disadvantages.

The value converter approach works best when you need to set a single property in an element based on the bound data object. It's easy, and it's automatically synchronized. If you make changes to the bound data object, the linked property is changed immediately.

Data triggers are similarly straightforward, but they support only extremely simple logic that tests for equality. For example, a data trigger can apply formatting that applies to products in a specific category, but it can't apply formatting that kicks in when the price is greater than a specific minimum value. The key advantage of data triggers is that you can use them to apply certain types of formatting and selection effects without writing any code.

Style selectors and template selectors are the most powerful options. They allow you to change multiple properties in the target element at once and change the way items are presented in the list. However, they introduce additional complexity. Also, you need to add code that reapplies your styles and templates if the bound data changes.

Evaluating Multiple Properties

So far, you've used binding expressions to convert one piece of source data into a single, formatted result. And although you can't change the second part of this equation (the result), you *can* create a binding that evaluates or combines the information in more than one source property, with a little craftiness.

The first trick is to replace your Binding object with a MultiBinding. Then you define the arrangement of bound properties by using the MultiBinding.StringFormat property. Here's an example that turns Joe and Smith into "Smith, Joe" and displays the result in a TextBlock:

```
<TextBlock>
  <TextBlock.Text>
    <MultiBinding StringFormat="{1}, {0}">
      <Binding Path="FirstName"></Binding>
      <Binding Path="LastName"></Binding>
    </MultiBinding>
  </TextBlock.Text>
</TextBlock>
```

You'll notice in this example that the two fields are used as is in the StringFormat property. Alternatively, you can use format strings to change it. For example, if you were combining a text value and a currency value with a MultiBinding, you might set StringFormat to "{0} costs {1:C}."

If you want to do anything more ambitious with the two source fields than simply stitching them together, you need the help of a value converter. You can use this technique to perform calculations (such as multiplying UnitPrice by UnitsInStock) or apply formatting that takes several details into consideration (such as highlighting all high-priced products in a specific category). However, your value converter must implement IMultiValueConverter rather than IValueConverter.

Here's an example in which a MultiBinding uses the UnitCost and UnitsInStock properties from the source object and combines them by using a value converter:

```
<TextBlock>Total Stock Value: </TextBlock>
<TextBox>
  <TextBox.Text>
    <MultiBinding Converter="{StaticResource ValueInStockConverter}">
      <Binding Path="UnitCost"></Binding>
      <Binding Path="UnitsInStock"></Binding>
    </MultiBinding>
  </TextBox.Text>
</TextBox>
```

The IMultiValueConverter interface defines similar Convert() and ConvertBack() methods as the IValueConverter interface. The main difference is that you're provided with an array of values rather than a single value. These values are placed in the same order that they're defined in your markup. Thus, in the previous example, you can expect UnitCost to appear first, followed by UnitsInStock.

Here's the code for the ValueInStockConverter:

```
public class ValueInStockConverter : IMultiValueConverter
{
    public object Convert(object[] values, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
    {
        // Return the total value of all the items in stock.
        decimal unitCost = (decimal)values[0];
        int unitsInStock = (int)values[1];
```

```
        return unitCost * unitsInStock;
    }

    public object[] ConvertBack(object value, Type[] targetTypes,
      object parameter, System.Globalization.CultureInfo culture)
    {
        throw new NotSupportedException();
    }
}
```

List Controls

String formatting and value converters are all you need to apply flexible formatting to individual bound values. But bound lists need a bit more. Fortunately, WPF provides no shortage of formatting choices. Most of these are built into the base ItemsControl class from which all list controls derive, so this is where your list-formatting exploration should start.

As you know, the ItemsControl class defines the basic functionality for controls that wrap a list of items. Those items can be entries in a list, nodes in a tree, commands in a menu, buttons in a toolbar, and so on. Figure 20-4 provides an at-a-glance overview of all the ItemsControl classes in WPF.

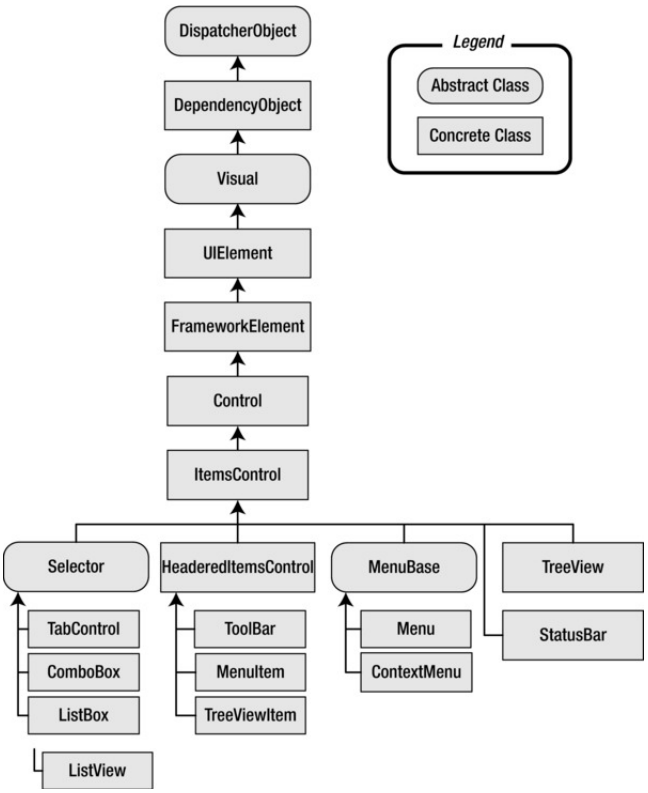

Figure 20-4. *Classes that derive from ItemsControl*

■ **Note** You'll notice that some item wrappers appear in the hierarchy of classes that derive from ItemsControl. For example, not only will you see the expected Menu and TreeView classes, but you'll also see MenuItem and TreeViewItem. That's because these classes have the ability to contain their own collection of items—that's what gives trees and menus their nested, hierarchical structure. On the other hand, you won't find ComboBoxItem or ListBoxItem in this list, because they don't need to hold a child collection of items and so don't derive from ItemsControl.

The ItemsControl defines the properties that support data binding and two key formatting features: styles and data templates. You'll explore both features in the following sections, and Table 20-3 gives you a quick overview of the properties that support them. The table is loosely organized from more-basic features to more-advanced ones, and it also mirrors the order you'll explore them in this chapter.

Table 20-3. *Formatting-Related Properties of the ItemsControl Class*

Name	Description
ItemsSource	The bound data source (the collection or DataView that you want to display in the list).
DisplayMemberPath	The property that you want to display for each data item. For a more sophisticated representation or to use a combination of properties, use the ItemTemplate instead.
ItemStringFormat	A .NET format string that, if set, will be used to format the text for each item. Usually, this technique is used to convert numeric or date values into a suitable display representation, exactly as the Binding.StringFormat property does.
ItemContainerStyle	A style that allows you to set the properties of the container that wraps each item. The container depends on the type of list (for example, it's ListBoxItem for the ListBox class and ComboBoxItem for the ComboBox class). These wrapper objects are created automatically as the list is filled.
ItemContainerStyleSelector	A StyleSelector that uses code to choose a style for the wrapper of each item in the list. This allows you to give different styles to different items in the list. You must create a custom StyleSelector yourself.
AlternationCount	The number of alternating sets in your data. For example, an AlternationCount of 2 alternates between two row styles, an AlternationCount of 3 alternates between three row styles, and so on.
ItemTemplate	A template that extracts the appropriate data out of your bound object and arranges it into the appropriate combination of controls.
ItemTemplateSelector	A DataTemplateSelector that uses code to choose a template for each item in the list. This allows you to give different templates to different items. You must create a custom DataTemplateSelector class yourself.
ItemsPanel	Defines the panel that's created to hold the items of the list. All the item wrappers are added to this container. Usually, a VirtualizingStackPanel is used with a vertical (top-to-bottom) orientation.

GroupStyle	If you're using grouping, this is a style that defines how each group should be formatted. When using grouping, the item wrappers (ListBoxItem, ComboBoxItem, and so on) are added in GroupItem wrappers that represent each group, and these groups are then added to the list. Grouping is demonstrated in Chapter 21.
GroupStyleSelector	A StyleSelector that uses code to choose a style for each group. This allows you to give different styles to different groups. You must create a custom StyleSelector yourself.

The next rung in the ItemsControl inheritance hierarchy is the Selector class, which adds a straightforward set of properties for determining (and setting) a selected item. Not all ItemsControl classes support selection. For example, selection doesn't have any meaning for the ToolBar or Menu, so these classes derive from ItemsControl but not Selector.

The properties that the Selector class adds include SelectedItem (the selected data object), SelectedIndex (the position of the selected item), and SelectedValue (the "value" property of the selected data object, which you designate by setting SelectedValuePath). Notice that the Selector class doesn't provide support for multiple selection—that's added to the ListBox through its SelectionMode and SelectedItems properties (which is essentially all the ListBox class adds to this model).

List Styles

For the rest of this chapter, you'll be concentrating on two features that are provided by all the WPF list controls: styles and data templates.

Out of these two tools, styles are simpler (and less powerful). In many cases, they allow you to add a bit of formatting polish. In the following sections, you'll see how styles let you format list items, apply alternating-row formatting, and apply conditional formatting according to the criteria you specify.

The ItemContainerStyle

In Chapter 11, you learned how styles allow you to reuse formatting with similar elements in different places. Styles play much the same role with lists—they allow you to apply a set of formatting characteristics to each of the individual items.

This is important, because WPF's data-binding system generates list item objects automatically. As a result, it's not so easy to apply the formatting you want to individual items. The solution is the ItemContainerStyle property. If the ItemContainerStyle is set, the list control will pass it down to each of its items, as the item is created. In the case of a ListBox control, each item is represented by a ListBoxItem object. (In a ComboBox, it's ComboBoxItem, and so on.) Thus, any style you apply with the ListBox.ItemContainerStyle property is used to set the properties of each ListBoxItem object.

Here's one of the simplest possible effects that you can achieve with the ListBoxItem. It applies a blue-gray background to each item. To make sure the individual items stand apart from each other (rather than having their backgrounds merge together), the style also adds some margin space:

```
<ListBox Name="lstProducts" Margin="5" DisplayMemberPath="ModelName">
  <ListBox.ItemContainerStyle>
    <Style>
      <Setter Property="ListBoxItem.Background" Value="LightSteelBlue" />
      <Setter Property="ListBoxItem.Margin" Value="5" />
      <Setter Property="ListBoxItem.Padding" Value="5" />
    </Style>
  </ListBox.ItemContainerStyle>
</ListBox>
```

On its own, this isn't terribly interesting. However, the style becomes a bit more polished with the addition of triggers. In the following example, property triggers change the background color and add a solid border when the ListBoxItem.IsSelected property becomes true. Figure 20-5 shows the result.

```
<ListBox Name="lstProducts" Margin="5" DisplayMemberPath="ModelName">
  <ListBox.ItemContainerStyle>
    <Style TargetType="{x:Type ListBoxItem}">
      <Setter Property="Background" Value="LightSteelBlue" />
      <Setter Property="Margin" Value="5" />
      <Setter Property="Padding" Value="5" />

      <Style.Triggers>
        <Trigger Property="IsSelected" Value="True">
          <Setter Property="Background" Value="DarkRed" />
          <Setter Property="Foreground" Value="White" />
          <Setter Property="BorderBrush" Value="Black" />
          <Setter Property="BorderThickness" Value="1" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </ListBox.ItemContainerStyle>
</ListBox>
```

For cleaner markup, this style uses the Style.TargetType property so that it can set properties without including the class name in each setter.

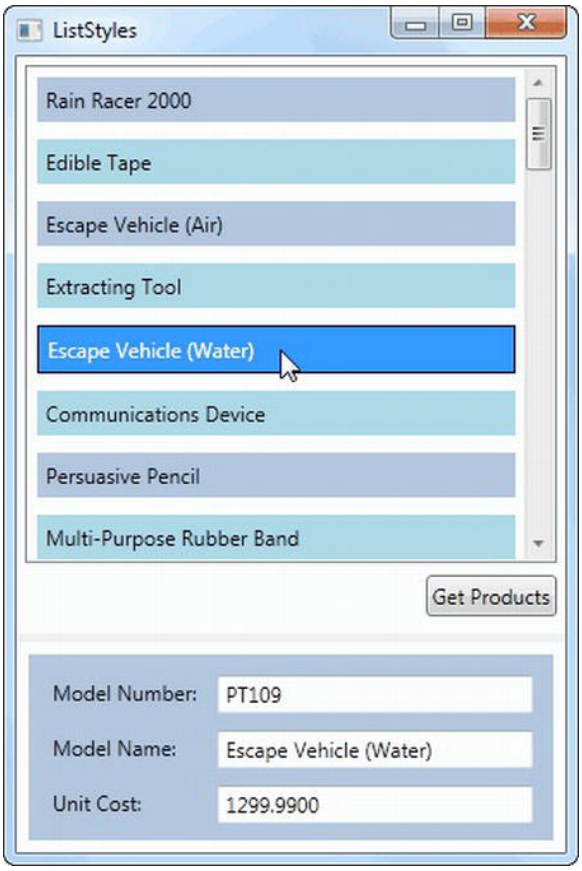

Figure 20-5. *Use style triggers to change the highlighting for the selected item.*

This use of triggers is particularly handy, because the ListBox doesn't provide any other way to apply targeted formatting to the selected item. In other words, if you don't use a style, you're stuck with the standard blue highlighting.

Later in this chapter, when you use data templates to completely revamp data lists, you'll once again rely on the ItemContainerStyle to change the selected item effect.

A ListBox with Check Boxes or Radio Buttons

The ItemContainerStyle is also important if you want to reach deep into a list control and change the control template that its items use. For example, you can use this technique to make every ListBoxItem display a radio button or a check box next to its item text.

Figure 20-6 and Figure 20-7 show two examples—one with a list filled with RadioButton elements (only one of which can be chosen at a time) and one with a list of CheckBox elements. The two solutions are similar, but the list with radio buttons is slightly easier to create.

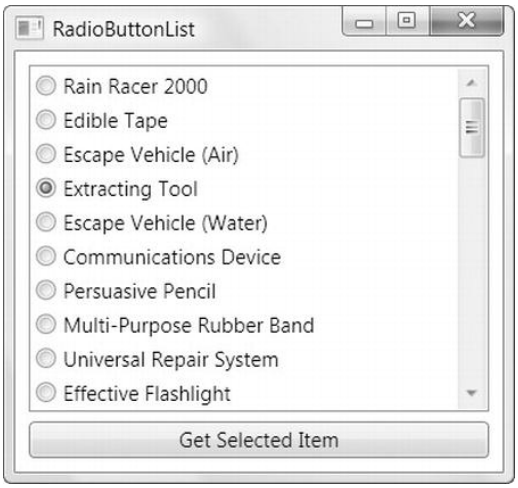

Figure 20-6. *A radio button list using a template*

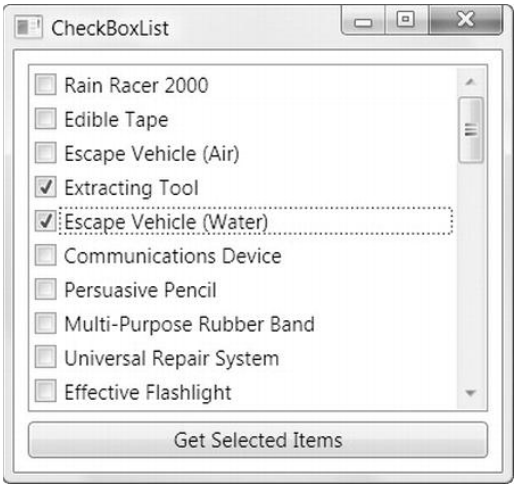

Figure 20-7. *A check box list using a template*

Note At first glance, using templates to change the ListBoxItem might seem like more work than it's worth. After all, it's easy enough to solve the problem by composition. All you need to do is fill a ScrollViewer with a series of CheckBox objects. However, this implementation doesn't provide the same programming model. There's no easy way to iterate through all the check boxes, and, more important, there's no way to use this implementation with data binding.

The basic technique in this example is to change the control template used as the container for each list item. You don't want to modify the ListBox.Template property, because this provides the template for the ListBox. Instead, you need to modify the ListBoxItem.Template property. Here's the template you need to wrap each item in a RadioButton element:

```
<ControlTemplate TargetType="{x:Type ListBoxItem}">
  <RadioButton Focusable="False" IsChecked="{Binding Path=IsSelected,
RelativeSource={RelativeSource TemplatedParent},Mode=TwoWay}">
    <ContentPresenter></ContentPresenter>
  </RadioButton>
</ControlTemplate>
```

This works because a RadioButton is a content control and can contain any content. Although you could use a binding expression to get the content, it's far more flexible to use the ContentPresenter element, as shown here. The ContentPresenter grabs whatever would ordinarily appear in the item, which might be property text (if you're using the ListBox.DisplayMemberPath property) or a more complex representation of the data (if you're using the ListBox.ItemTemplate property).

The real trick is the binding expression for the RadioButton.IsChecked property. This expression retrieves the value of the ListBoxItem.IsSelected property by using the Binding.RelativeSource property. That way, when you click a RadioButton to select it, the corresponding ListBoxItem is marked as selected. At the same time, all other items are deselected. This binding expression also works in the other direction, which means you can set the selection in code and the right RadioButton will be filled in.

To complete this template, you need to set the RadioButton.Focusable property to false. Otherwise, you'll be able to tab to the currently selected ListBoxItem (which is focusable) and then into the RadioButton itself, which doesn't make much sense.

To set the ListBoxItem.Template property, you need a style rule that can dig down to the right level. Thanks to the ItemContainerStyle property, this part is easy:

```
<Window.Resources>
  <Style x:Key="RadioButtonListStyle" TargetType="{x:Type ListBox}">
    <Setter Property="ItemContainerStyle">
      <Setter.Value>
        <Style TargetType="{x:Type ListBoxItem}" >
          <Setter Property="Margin" Value="2" />
          <Setter Property="Template">
            <Setter.Value>
              <ControlTemplate TargetType="{x:Type ListBoxItem}">
                <RadioButton Focusable="False"
                 IsChecked="{Binding Path=IsSelected, Mode=TwoWay,
                             RelativeSource={RelativeSource TemplatedParent} }">
                  <ContentPresenter></ContentPresenter>
                </RadioButton>
              </ControlTemplate>
            </Setter.Value>
          </Setter>
        </Style>
      </Setter.Value>
    </Setter>
  </Style>
</Window.Resources>
```

Although you could set the ListBox.ItemContainerStyle property directly, this example factors it out one more level. The style that sets the ListBoxItem.Control template is wrapped in another style that applies this style to the ListBox.ItemContainerStyle property. This makes the template reusable, allowing you to connect it to as many ListBox objects as you want.

```
<ListBox Style="{StaticResource RadioButtonListStyle}" Name="lstProducts"
 DisplayMemberPath="ModelName">
```

You could also use the same style to adjust other properties of the ListBox.

Creating a ListBox that shows check boxes is just as easy. In fact, you have to make only two changes. First, replace the RadioButton element with an identical CheckBox element. Then change the ListBox.SelectionMode property to allow simple multiple selection. Now the user can check as many or as few items as desired.

Here's the style rule that transforms an ordinary ListBox into a list of check boxes:

```
<Style x:Key="CheckBoxListStyle" TargetType="{x:Type ListBox}">
  <Setter Property="SelectionMode" Value="Multiple"></Setter>
  <Setter Property="ItemContainerStyle">
    <Setter.Value>
      <Style TargetType="{x:Type ListBoxItem}" >
        <Setter Property="Margin" Value="2" />
        <Setter Property="Template">
          <Setter.Value>
            <ControlTemplate TargetType="{x:Type ListBoxItem}">
              <CheckBox Focusable="False"
               IsChecked="{Binding Path=IsSelected, Mode=TwoWay,
                           RelativeSource={RelativeSource TemplatedParent} }">
                <ContentPresenter></ContentPresenter>
              </CheckBox>
            </ControlTemplate>
          </Setter.Value>
        </Setter>
      </Style>
    </Setter.Value>
  </Setter>
</Style>
```

Alternating Item Style

One common way to format a list is to use alternating row formatting—in other words, a set of formatting characteristics that distinguishes every second item in a list. Often, alternating rows are given subtly different background colors so that the rows are clearly separated, as shown in Figure 20-8.

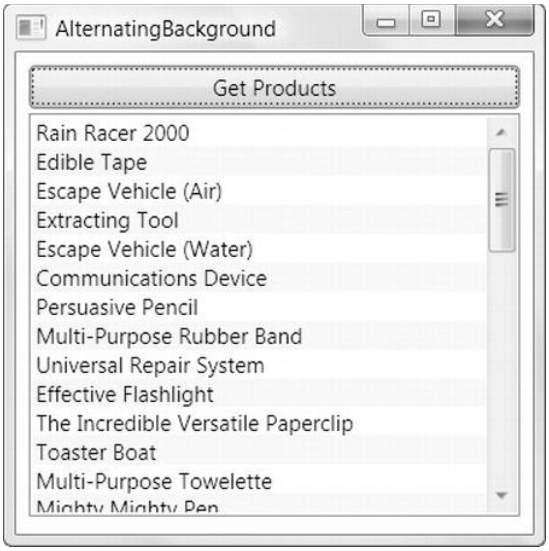

Figure 20-8. *Alternating-row highlighting*

WPF has built-in support for alternating items through two properties: AlternationCount and AlternationIndex.

AlternationCount is the number of items that form a sequence, after which the style alternates. By default, AlternationCount is set to 0, and alternating formatting isn't used. If you set AlternationCount to 1, the list will alternate after every item, which allows you to apply the even-odd formatting pattern shown in Figure 20-8.

Every ListBoxItem is given an AlternationIndex, which allows you to determine how it's numbered in the sequence of alternating items. Assuming you've set AlternationCount to 2, the first ListBoxItem gets an AlternationIndex of 0, the second gets an AlternationIndex of 1, the third gets an AlternationIndex of 0, the fourth gets an AlternationIndex of 1, and so on. The trick is to use a trigger in your ItemContainerStyle that checks the AlternationIndex value and varies the formatting accordingly.

For example, the ListBox control shown here gives alternate items a slightly different background color (unless the item is selected, in which case the higher-priority trigger for ListBoxItem.IsSelected wins out):

```
<ListBox Name="lstProducts" Margin="5" DisplayMemberPath="ModelName"
 AlternationCount=”2”>
  <ListBox.ItemContainerStyle>
    <Style TargetType="{x:Type ListBoxItem}">
      <Setter Property="Background" Value="LightSteelBlue" />
      <Setter Property="Margin" Value="5" />
      <Setter Property="Padding" Value="5" />
        <Style.Triggers>
          <Trigger Property="ItemsControl.AlternationIndex" Value="1">
            <Setter Property="Background" Value="LightBlue" />
          </Trigger>
          <Trigger Property="IsSelected" Value="True">
            <Setter Property="Background" Value="DarkRed" />
```

```
            <Setter Property="Foreground" Value="White" />
            <Setter Property="BorderBrush" Value="Black" />
            <Setter Property="BorderThickness" Value="1" />
          </Trigger>
        </Style.Triggers>
      </Style>
    </ListBox.ItemContainerStyle>
</ListBox>
```

You'll notice that AlternationIndex is an attached property that's defined by the ListBox class (or, technically, the ItemsControl class that it derives from). It's not defined in the ListBoxItem class, so when you use it in a style trigger, you need to specify the class name.

Interestingly, alternating items don't need to be every second item. Instead, you can create more-complex alternating formatting that alternates in a sequence of three or more. For example, to use three groups, set AlternationCount to 3, and write triggers for any of the three possible AlternationIndex values (0, 1, or 2). In the list, items 1, 4, 7, 10, and so on, will have an AlternationIndex of 0. Items 2, 5, 8, 11, and so on, get an AlternationIndex of 1. And finally, items 3, 6, 9, 12, and so on, get an AltenationIndex of 2.

Style Selectors

You've now seen how to vary the style based on the selection state of the item or its position in the list. However, you might want to use a number of other conditions—criteria that depend on your data rather than the ListBoxItem container that holds it.

To deal with this situation, you need a way to give different items completely different styles. Unfortunately, there's no way to do this declaratively. Instead, you need to build a specialized class that derives from StyleSelector. This class has the responsibility of examining each data item and choosing the appropriate style. This work is performed in the SelectStyle() method, which you must override.

Here's a rudimentary selector that chooses between two styles:

```
public class ProductByCategoryStyleSelector : StyleSelector
{
    public override Style SelectStyle(object item, DependencyObject container)
    {
        Product product = (Product)item;
        Window window = Application.Current.MainWindow;

        if (product.CategoryName == "Travel")
        {
            return (Style)window.FindResource("TravelProductStyle");
        }
        else
        {
            return (Style)window.FindResource("DefaultProductStyle");
        }
    }
}
```

In this example, products that are in the Travel category get one style, while all other products get another. In this example, both styles you want to use must be defined in the Resources collection of the window, with the key names TravelProductStyle and DefaultProductStyle.

This style selector works, but it's not perfect. One problem is that your code depends on details that are in the markup, which means there's a dependency that isn't enforced at compile time and could easily be disrupted (for example, if you give your styles the wrong resource keys). The other problem is that this style selector hard-codes the value it's looking for (in this case, the category name), which limits reuse.

A better idea is to create a style selector that uses one or more properties to allow you to specify some of these details, such as the criteria you're using to evaluate your data items and the styles you want to use. The following style selector is still quite simple but extremely flexible. It's able to examine any data object, look for a given property, and compare that property against another value to choose between two styles. The property, property value, and styles are all specified as properties. The SelectStyle() method uses reflection to find the right property in a manner similar to the way data bindings work when digging out bound values.

Here's the complete code:

```
public class SingleCriteriaHighlightStyleSelector : StyleSelector
{
    public Style DefaultStyle
    {
        get; set;
    }

    public Style HighlightStyle
    {
        get; set;
    }

    public string PropertyToEvaluate
    {
        get; set;
    }

    public string PropertyValueToHighlight
    {
        get; set;
    }

    public override Style SelectStyle(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 HighlightStyle;
        }
        else
```

```
        {
            return DefaultStyle;
        }
    }
}
```

To make this work, you'll need to create the two styles you want to use, and you'll need to create and initialize an instance of the SingleCriteriaHighlightStyleSelector.

Here are two similar styles, which are distinguished only by the background color and the use of bold formatting:

```
<Window.Resources>
  <Style x:Key="DefaultStyle" TargetType="{x:Type ListBoxItem}">
    <Setter Property="Background" Value="LightYellow" />
    <Setter Property="Padding" Value="2" />
  </Style>

  <Style x:Key="HighlightStyle" TargetType="{x:Type ListBoxItem}">
    <Setter Property="Background" Value="LightSteelBlue" />
    <Setter Property="FontWeight" Value="Bold" />
    <Setter Property="Padding" Value="2" />
  </Style>
</Window.Resources>
```

When you create the SingleCriteriaHighlightStyleSelector, you point it to these two styles. You can also create the SingleCriteriaHighlightStyleSelector as a resource (which is useful if you want to reuse it in more than one place), or you can define it inline in your list control, as in this example:

```
<ListBox Name="lstProducts" HorizontalContentAlignment="Stretch">
  <ListBox.ItemContainerStyleSelector>
    <local:SingleCriteriaHighlightStyleSelector
      DefaultStyle="{StaticResource DefaultStyle}"
      HighlightStyle="{StaticResource HighlightStyle}"
      PropertyToEvaluate="CategoryName"
      PropertyValueToHighlight="Travel"
    >
    </local:SingleCriteriaHighlightStyleSelector>
  </ListBox.ItemContainerStyleSelector>
</ListBox>
```

Here, the SingleCriteriaHighlightStyleSelector looks for a Category property in the bound data item and uses the HighlightStyle if it contains the text *Travel*. Figure 20-9 shows the result.

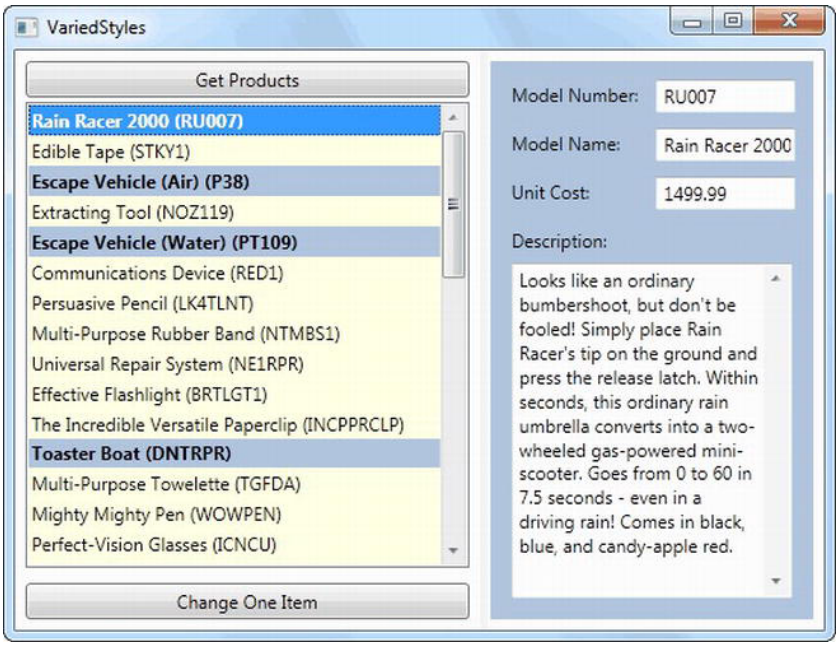

Figure 20-9. *A list with two item styles*

The style selection process is performed once, when you first bind the list. This is a problem if you're displaying editable data and it's possible for an edit to move the data item from one style category to another. In this situation, you need to force WPF to reapply the styles, and there's no graceful way to do it. The brute-force approach is to remove the style selector by setting the ItemContainerStyleSelector property to null and then to reassign it:

```
StyleSelector selector = lstProducts.ItemContainerStyleSelector;
lstProducts.ItemContainerStyleSelector = null;
lstProducts.ItemContainerStyleSelector = selector;
```

You may choose to run this code automatically in response to certain changes by handling events such as PropertyChanged (which is raised by all classes that implement INotifyPropertyChanged, including Product), DataTable.RowChanged (if you're using the ADO.NET data objects), and, more generically, Binding.SourceUpdated (which fires only when Binding.NotifyOnSourceUpdated is true). When you reassign the style selector, WPF examines and updates every item in the list—a process that's quick for small or medium-size lists.

Data Templates

Styles give you some basic formatting abilities, but they don't address the most significant limitation of the lists you've seen so far: no matter how you tweak the ListBoxItem, it's only a ListBoxItem, not a more capable combination of elements. And because each ListBoxItem supports just a single bound field (as set through the DisplayMemberPath property), there's no room to make a rich list that incorporates multiple fields or images.

However, WPF does have another tool that can break out of this rather limiting box, allowing you to use a combination of properties from the bound object and lay them out in a specific way or to display a visual representation that's more sophisticated than a simple string. That tool is the data template.

A *data template* is a chunk of XAML markup that defines how a bound data object should be displayed. Two types of controls support data templates:

- Content controls support data templates through the ContentTemplate property. The content template is used to display whatever you've placed in the Content property.
- List controls (controls that derive from ItemsControl) support data templates through the ItemTemplate property. This template is used to display each item from the collection (or each row from a DataTable) that you've supplied as the ItemsSource.

The list-based template feature is actually based on content control templates. That's because each item in a list is wrapped by a content control, such as ListBoxItem for the ListBox, ComboBoxItem for the ComboBox, and so on. Whatever template you specify for the ItemTemplate property of the list is used as the ContentTemplate of each item in the list.

So, what can you put inside a data template? It's actually quite simple. A data template is an ordinary block of XAML markup. Like any other block of XAML markup, the template can include any combination of elements. It should also include one or more data-binding expressions that pull out the information that you want to display. (After all, if you don't include any data-binding expressions, each item in the list will appear the same, which isn't very helpful.)

The best way to see how a data template works is to start with a basic list that doesn't use them. For example, consider this list box, which was shown previously:

```
<ListBox Name="lstProducts" DisplayMemberPath="ModelName"></ListBox>
```

You can get the same effect with this list box that uses a data template:

```
<ListBox Name="lstProducts">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding Path=ModelName}"></TextBlock>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

When the list is bound to the collection of products (by setting the ItemsSource property), a single ListBoxItem is created for each Product. The ListBoxItem.Content property is set to the appropriate Product object, and the ListBoxItem.ContentTemplate is set to the data template shown earlier, which extracts the value from the Product.ModelName property and displays it in a TextBlock.

So far, the results are underwhelming. But now that you've switched to a data template, there's no limit to how you can creatively present your data. Here's an example that wraps each item in a rounded border, shows two pieces of information, and uses bold formatting to highlight the model number:

```
<ListBox Name="lstProducts" HorizontalContentAlignment="Stretch">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <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>
  </ListBox.ItemTemplate>
</ListBox>
```

When this list is bound, a separate Border object is created for each product. Inside the Border element is a Grid with two pieces of information, as shown in Figure 20-10.

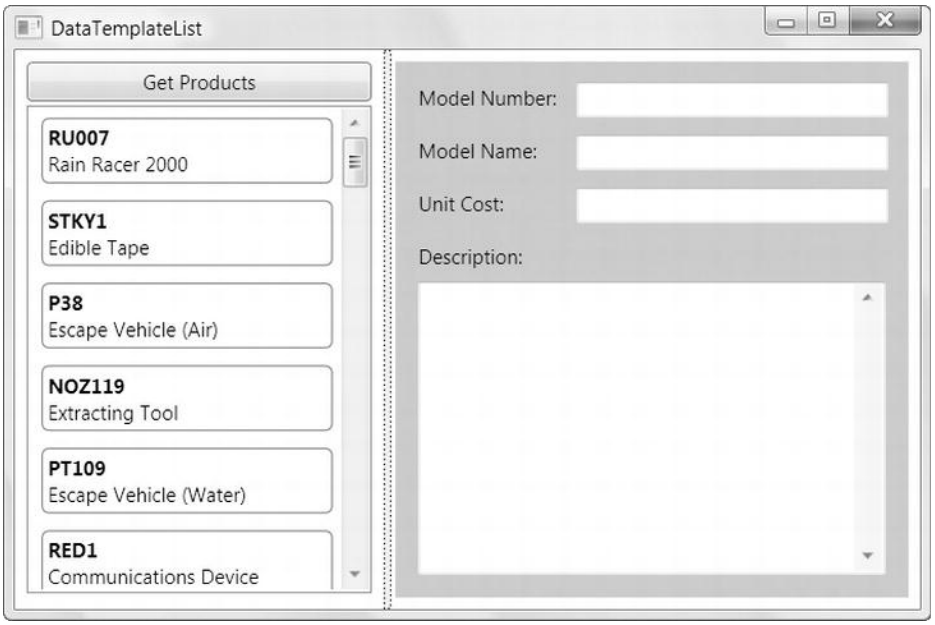

Figure 20-10. *A list that uses a data template*

■ **Tip** When using Grid objects to lay out individual items in a list, you may want to use the SharedSizeGroup property described in Chapter 3. You can apply the SharedSizeGroup property (with a descriptive group name) to individual rows or columns to ensure that those rows and columns are made the same size for every item. Chapter 22 includes an example that uses this approach to build a rich view for the ListView that combines text and image content.

Separating and Reusing Templates

Like styles, templates are often declared as a window or application resource rather than defined in the list where you use them. This separation is often clearer, especially if you use long, complex templates or multiple templates in the same control (as described in the next section). It also gives you the ability to reuse your templates in more than one list or content control if you want to present your data the same way in different places in your user interface.

To make this work, all you need to do is to define your data template in a resources collection and give it a key name. Here's an example that extracts the template shown in the previous example:

```
<Window.Resources>
  <DataTemplate x:Key="ProductDataTemplate">
    <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>
</Window.Resources>
```

Now you can add your data template to the list using a StaticResource reference:

```
<ListBox Name="lstProducts" HorizontalContentAlignment="Stretch"
 ItemTemplate="{StaticResource ProductDataTemplate}"></ListBox>
```

You can use another interesting trick if you want to reuse the same data template in different types of controls automatically. You can set the DataTemplate.DataType property to identify the type of bound data for which your template should be used. For example, you could alter the previous example by removing the key and specifying that this template is intended for bound Product objects, no matter where they appear:

```
<Window.Resources>
  <DataTemplate DataType="{x:Type local:Product}">
  </DataTemplate>
</Window.Resources>
```

This assumes that you've defined an XML namespace prefix named *local* and mapped it to your project namespace.

Now this template will be used with any list or content control in this window that's bound to Product objects. You don't need to specify the ItemTemplate setting.

■ **Note** Data templates don't require data binding. In other words, you don't need to use the ItemsSource property to fill a template list. In the previous examples, you're free to add Product objects declaratively (in your XAML markup) or programmatically (by calling the ListBox.Items.Add() method). In both cases, the data template works in the same way.

Using More Advanced Templates

Data templates can be remarkably self-sufficient. Along with basic elements such as the TextBlock and data-binding expressions, they can also use more-sophisticated controls, attach event handlers, convert data to different representations, use animations, and so on.

It's worth considering a couple of quick examples that show how powerful data templates are. First, you can use value converter objects in your data binding to convert your data to a more useful representation. The following example uses the ImagePathConverter demonstrated earlier to show the image for each product in the list:

```
<Window.Resources>
  <local:ImagePathConverter x:Key="ImagePathConverter"></local:ImagePathConverter>
  <DataTemplate x:Key="ProductTemplate">
    <Border Margin="5" BorderThickness="1" BorderBrush="SteelBlue"
     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" Text="{Binding Path=ModelName}"></TextBlock>
        <Image Grid.Row="2" Grid.RowSpan="2" Source=
"{Binding Path=ProductImagePath, Converter={StaticResource ImagePathConverter}}">
        </Image>
      </Grid>
    </Border>
  </DataTemplate>
</Window.Resources>
```

Although this markup doesn't involve anything exotic, the result is a much more interesting list (see Figure 20-11).

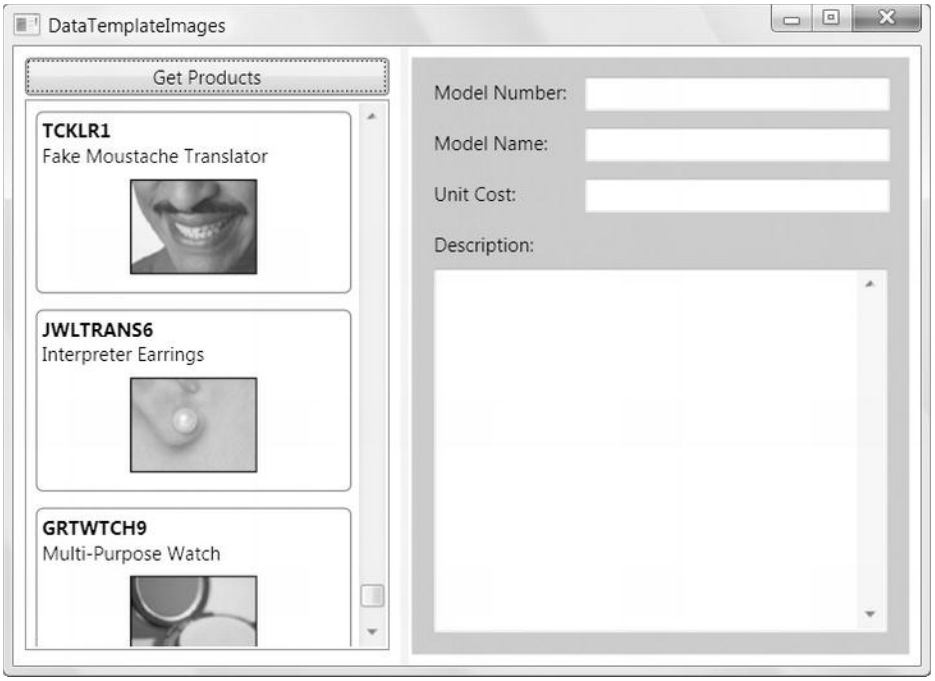

Figure 20-11. *A list with image content*

Another useful technique is to place controls directly inside a template. For example, Figure 20-12 shows a list of categories. Next to each category is a View button that you can use to launch another window with just the matching products in that category.

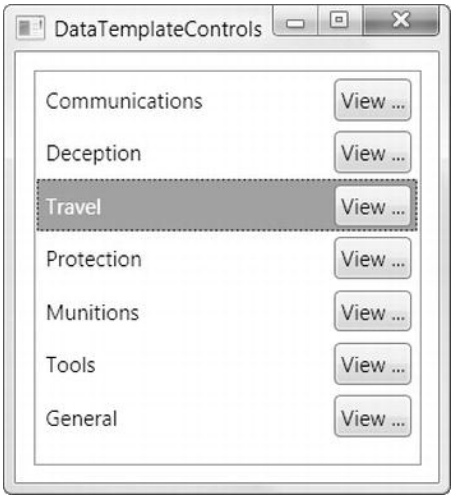

Figure 20-12. *A list with button controls*

The trick in this example is handling the button clicks. Obviously, all of the buttons will be linked to the same event handler, which you define inside the template. However, you need to determine which item was clicked from the list. One solution is to store some extra identifying information in the Tag property of the button, as shown here:

```
<DataTemplate>
  <Grid Margin="3">
    <Grid.ColumnDefinitions>
      <ColumnDefinition></ColumnDefinition>
      <ColumnDefinition Width="Auto"></ColumnDefinition>
    </Grid.ColumnDefinitions>

    <TextBlock Text="{Binding Path=CategoryName}"></TextBlock>
    <Button Grid.Column="2" HorizontalAlignment="Right" Padding="2"
      Click="cmdView_Clicked" Tag="{Binding Path=CategoryID}">View ...</Button>
  </Grid>
</DataTemplate>
```

You can then retrieve the Tag property in the cmdView_Clicked event handler:

```
private void cmdView_Clicked(object sender, RoutedEventArgs e)
{
    Button cmd = (Button)sender;
    int categoryID = (int)cmd.Tag;
    ...
}
```

You can use this information to take another action. For example, you might launch another window that shows products and pass the CategoryID value to that window, which can then use filtering to show only the products in that category. (One easy way to implement filtering is with data views, as described in Chapter 21.)

If you want all the information about the selected data item, you can grab the entire data object by leaving out the Path property when you define the binding:

```
<Button HorizontalAlignment="Right" Padding="1"
  Click="cmdView_Clicked" Tag="{Binding}">View ...</Button>
```

Now your event handler will receive the Product object (if you're binding a collection of Products). If you're binding to a DataTable, you'll receive a DataRowView object instead, which you can use to retrieve all the field values exactly as you would with a DataRow object.

Passing the entire object has another advantage: it makes it easier to update the list selection. In the current example, it's possible to click a button in any item, regardless of whether that item is selected. This is potentially confusing, because the user could select one item and click the View button of another item. When the user returns to the list window, the first item remains selected even though the second item was the one that was used by the previous operation. To remove the possibility for confusion, it's a good idea to move the selection to the new list item when the View button is clicked, as shown here:

```
Button cmd = (Button)sender;
Product product = (Product)cmd.Tag;
lstCategories.SelectedItem = product;
```

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 the 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 by 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).

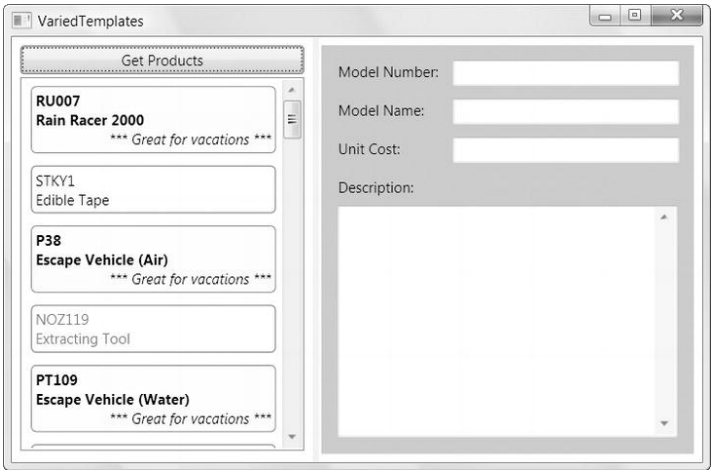

Figure 20-13. *A list with two data templates*

■ **Tip** One disadvantage of 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.

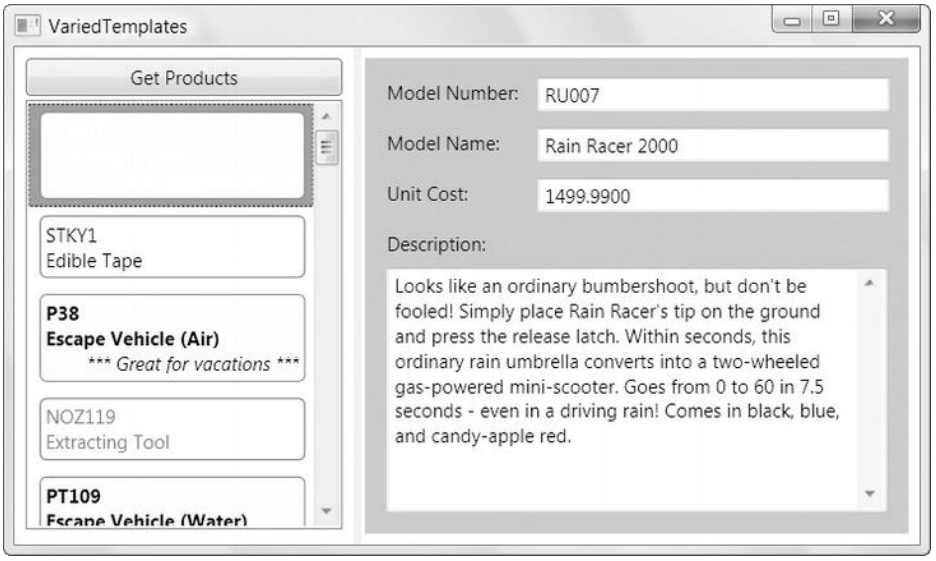

Figure 20-14. *Unreadable text in a highlighted item*

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>
```

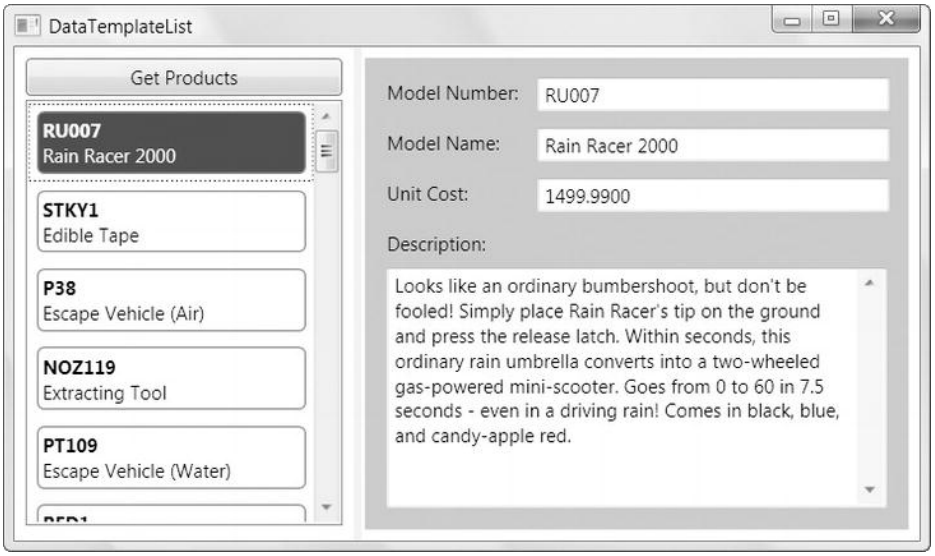

Figure 20-15. *Highlighting a selected item*

SELECTION AND SNAPSTODEVICEPIXELS

You should make one other change to ensure that 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, you could 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 products 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.

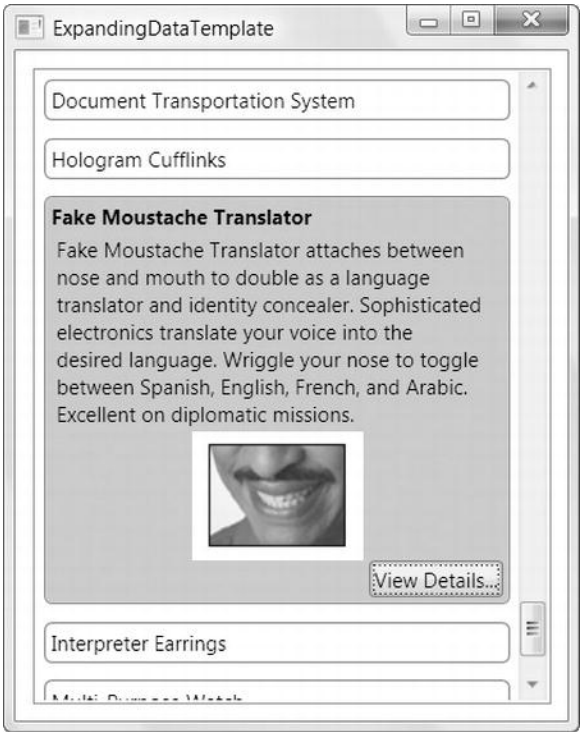

Figure 20-16. *Expanding a selected item*

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 by 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 by 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 used 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 scrollbar. 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.

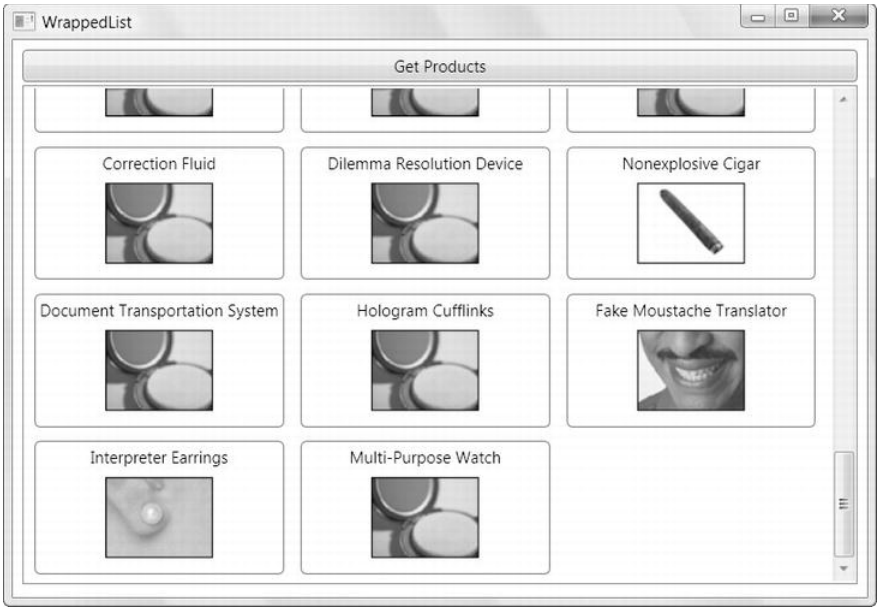

Figure 20-17. *Tiling items in the display area of a list*

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 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.

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 list 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 from which 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.

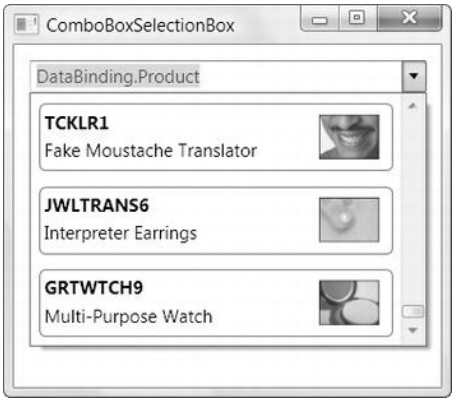

Figure 20-19. *An editable ComboBox that uses templates*

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 it 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.