CHAPTER 22

Lists, Trees, and Grids

So far, you've learned a wide range of techniques and tricks for using WPF data binding to display information in the form you need. Along the way, you've seen many examples that revolve around the lowly ListBox control.

Thanks to the extensibility provided by styles, data templates, and control templates, even the ListBox (and its similarly equipped sibling, the ComboBox) can serve as a remarkably powerful tool for displaying data in a variety of ways. However, some types of data presentation would be difficult to implement with the ListBox alone. Fortunately, WPF has a few rich data controls that fill in the blanks, including the following:

ListView: The ListView derives from the plain-vanilla ListBox. It adds support for column-based display and the ability to switch quickly between different "views," or display modes, without requiring you to rebind the data and rebuild the list.

TreeView: The TreeView is a hierarchical container, which means you can create a multilayered data display. For example, you could create a TreeView that shows category groups in its first level and shows the related products under each category node.

DataGrid: The DataGrid is WPF's most full-featured data display tool. It divides your data into a grid of columns and rows, like the ListView, but has additional formatting features (such as the ability to freeze columns and style individual rows), and it supports in-place data editing.

In this chapter, you'll look at these three key controls.

The ListView

The ListView is a specialized list class that's designed for displaying different *views* of the same data. The ListView is particularly useful if you need to build a multicolumn view that displays several pieces of information about each data item.

The ListView derives from the ListBox class and extends it with a single detail: the View property. The View property is yet another extensibility point for creating rich list displays. If you don't set the View property, the ListView behaves just like its lesser-powered ancestor, the ListBox. However, the ListView becomes much more interesting when you supply a view object that indicates how data items should be formatted and styled.

Technically, the View property points to an instance of any class that derives from ViewBase (which is an abstract class). The ViewBase class is surprisingly simple; in fact, it's little more than a package that binds together two styles. One style applies to the ListView control (and is referenced by the DefaultStyleKey property), and the other style applies to the items in the ListView (and is referenced by the ItemContainerDefaultStyleKey property). The DefaultStyleKey and ItemContainerDefaultStyleKey properties don't actually provide the style; instead, they return a ResourceKey object that points to it.

At this point, you might wonder why you need a View property—after all, the ListBox already offers powerful data template and styling features (as do all classes that derive from ItemsControl). Ambitious developers can rework the visual appearance of the ListBox by supplying a different data template, layout panel, and control template.

In truth, you don't need a ListView class with a View property in order to create customizable multicolumned lists. In fact, you could achieve much the same thing on your own by using the template and styling features of the ListBox. However, the View property is a useful abstraction. Here are some of its advantages:

Reusable views: The ListView separates all the view-specific details into one object. That makes it easier to create views that are data-independent and can be used on more than one list.

Multiple views: The separation between the ListView control and the View objects also makes it easier to switch between multiple views with the same list. (For example, you use this technique in Windows Explorer to get a different perspective on your files and folders.) You could build the same feature by dynamically changing templates and styles, but it's easier to have just one object that encapsulates all the view details.

Better organization: The view object wraps two styles: one for the root ListView control and one that applies to the individual items in the list. Because these styles are packaged together, it's clear that these two pieces are related and may share certain details and interdependencies. For example, this makes a lot of sense for a column-based ListView, because it needs to keep its column headers and column data lined up.

Using this model, there's a great potential to create a number of useful prebuilt views that all developers can use. Unfortunately, WPF currently includes just one view object: the GridView. Although the GridView is extremely useful for creating multicolumn lists, you'll need to create your own custom view if you have other needs. The following sections show you how to do both.

■ **Note** The GridView is a good choice if you want to show a configurable data display, and you want a grid-styled view to be one of the user's options. But if you want a grid that supports advanced styling, selection, or editing, you'll need to step up to the full-fledged DataGrid control described later in this chapter.

Creating Columns with the GridView

The GridView is a class that derives from ViewBase and represents a list view with multiple columns. You define those columns by adding GridViewColumn objects to the GridView.Columns collection.

Both GridView and GridViewColumn provide a small set of useful methods that you can use to customize the appearance of your list. To create the simplest, most straightforward list (which resembles

the details view in Windows Explorer), you need to set just two properties for each GridViewColumn: Header and DisplayMemberBinding. The Header property supplies the text that's placed at the top of the column. The DisplayMemberBinding property contains a binding that extracts the piece of information you want to display from each data item.

Figure 22-1 shows a straightforward example with three columns of information about a product.

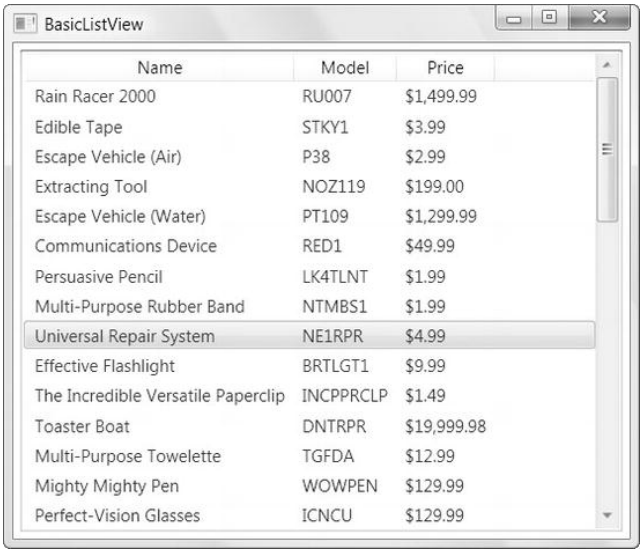

Figure 22-1. *A grid-based ListView*

Here's the markup that defines the three columns used in this example:

```
<ListView Margin="5" Name="lstProducts">
  <ListView.View>
    <GridView>
      <GridView.Columns>
        <GridViewColumn Header="Name"
         DisplayMemberBinding="{Binding Path=ModelName}" />
        <GridViewColumn Header="Model"
         DisplayMemberBinding="{Binding Path=ModelNumber}" />
        <GridViewColumn Header="Price" DisplayMemberBinding=
"{Binding Path=UnitCost, StringFormat={}{0:C}}" />
      </GridView.Columns>
    </GridView>
  </ListView.View>
</ListView>
```

This example has a few important points worth noticing. First, none of the columns has a hard-coded size. Instead, the GridView sizes its columns just large enough to fit the widest visible item (or the column header, if it's wider), which makes a lot of sense in the flow layout world of WPF. (Of course, this leaves you in a bit of trouble if you have huge column values. In this case, you may choose to wrap your text, as described in the upcoming "Cell Templates" section.)

Also, notice how the DisplayMemberBinding property is set by using a full-fledged binding expression, which supports all the tricks you learned about in Chapter 20, including string formatting and value converters.

Resizing Columns

Initially, the GridView makes each column just wide enough to fit the largest visible value. However, you can easily resize any column by clicking and dragging the edge of the column header. Or you can double-click the edge of the column header to force the GridViewColumn to resize itself based on whatever content is currently visible. For example, if you scroll down the list and find an item that's truncated because it's wider than the column, just double-click the right edge of that column's header. The column will automatically expand itself to fit.

For more micromanaged control over column size, you can set a specific width when you declare the column:

```
<GridViewColumn Width="300" ... />
```

This simply determines the initial size of the column. It doesn't prevent the user from resizing the column by using either of the techniques described previously. Unfortunately, the GridViewColumn class doesn't define properties such as MaxWidth and MinWidth, so there's no way to constrain how a column can be resized. Your only option is to supply a new template for the GridViewColumn's header if you want to disable resizing altogether.

■ **Note** The user can also reorder columns by dragging a header to a new position.

Using Cell Templates

The GridViewColumn.DisplayMemberBinding property isn't the only option for showing data in a cell. Your other choice is the CellTemplate property, which takes a data template. This is exactly like the data templates you learned about in Chapter 20, except it applies to just one column. If you're ambitious, you can give each column its own data template.

Cell templates are a key piece of the puzzle when customizing the GridView. One feature that they allow is text wrapping. Ordinarily, the text in a column is wrapped in a single-line TextBlock. However, it's easy to change this detail by using a data template of your own devising:

```
<GridViewColumn Header="Description" Width="300">
  <GridViewColumn.CellTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding Path=Description}" TextWrapping="Wrap"></TextBlock>
    </DataTemplate>
  </GridViewColumn.CellTemplate>
</GridViewColumn>
```

Notice that in order for the wrapping to have an effect, you need to constrain the width of the column by using the Width property. If the user resizes the column, the text will be rewrapped to fit. You *don't* want to constrain the width of the TextBlock, because that would ensure that your text is limited to a single specific size, no matter how wide or narrow the column becomes.

The only limitation in this example is that the data template needs to bind explicitly to the property you want to display. For that reason, you can't create a template that enables wrapping and reuse it for

every piece of content you want to wrap. Instead, you need to create a separate template for each field. This isn't a problem in this simple example, but it's annoying if you create a more complex template that you would like to apply to other lists (for example, a template that converts data to an image and displays it in an Image element, or a template that uses a TextBox control to allow editing). There's no easy way to reuse any template on multiple columns; instead, you'll be forced to cut and paste the template, and then modify the binding.

■ **Note** It would be nice if you could create a data template that uses the DisplayMemberBinding property. That way, you could use DisplayMemberBinding to extract the specific property you want and use CellTemplate to format that content into the correct visual representation. Unfortunately, this just isn't possible. If you set both DisplayMember and CellTemplate, the GridViewColumn uses the DisplayMember property to set the content for the cell and ignores the template altogether.

Data templates aren't limited to tweaking the properties of a TextBlock. You can also use data templates to supply completely different elements. For example, the following column uses a data template to show an image. The ProductImagePath converter (shown in Chapter 20) helps by loading the corresponding image file from the file system.

```
<GridViewColumn Header="Picture" >
  <GridViewColumn.CellTemplate>
    <DataTemplate>
      <Image Source=
"{Binding Path=ProductImagePath,Converter={StaticResource ImagePathConverter}}">
      </Image>
    </DataTemplate>
  </GridViewColumn.CellTemplate>
</GridViewColumn>
```

Figure 22-2 shows a ListView that uses both templates to show wrapped text and a product image.

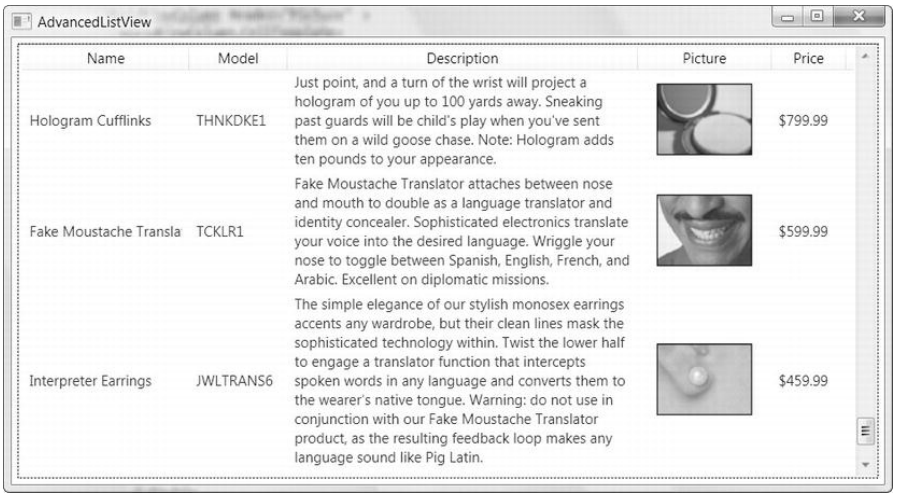

Figure 22-2. *Columns that use templates*

■ **Tip** When creating a data template, you have the choice of defining it inline (as in the previous two examples) or referring to a resource that's defined elsewhere. Because column templates can't be reused for different fields, it's usually clearest to define them inline.

As you learned in Chapter 20, you can vary templates so that different data items get different templates. To do this, you need to create a template selector that chooses the appropriate template based on the properties of the data object at that position. To use this feature, create your selector, and use it to set the GridViewColumn.CellTemplateSelector property. For a full template selector example, see Chapter 20.

CUSTOMIZING COLUMN HEADERS

So far, you've seen how to customize the appearance of the values in every cell. However, you haven't done anything to fine-tune the column headers. If the standard gray boxes don't excite you, you'll be happy to find out that you can change the content and appearance of the column headers just as easily as the column values. In fact, you can use several approaches.

If you want to keep the gray column header boxes but you want to fill them with your own content, you can simply set the GridViewColumn.Header property. The previous examples have the Header property using ordinary text, but you can supply an element instead. Use a StackPanel that wraps a TextBlock and Image to create a fancy header that combines text and image content.

If you want to fill the column headers with your own content, but you don't want to specify this content separately for each column, you can use the GridViewColumn.HeaderTemplate property to define a data template. This data template binds to whatever object you've specified in the GridViewColumn.Header property and presents it accordingly.

If you want to reformat a specific column header, you can use the GridViewColumn.HeaderContainerStyle property to supply a style. If you want to reformat all the column headers in the same way, use the GridView.ColumnHeaderContainerStyle property instead.

If you want to completely change the appearance of the header (for example, replacing the gray box with a rounded blue border), you can supply a completely new control template for the header. Use GridViewColumn.HeaderTemplate to change a specific column, or use GridView.ColumnHeaderTemplate to change them all in the same way. You can even use a template selector to choose the correct template for a given header by setting the GridViewColumn.HeaderTemplateSelector or GridView.ColumnHeaderTemplateSelector property.

Creating a Custom View

If the GridView doesn't meet your needs, you can create your own view to extend the ListView's capabilities. Unfortunately, it's far from straightforward.

To understand the problem, you need to know a little more about the way a view works. Views do their work by overriding two protected properties: DefaultStyleKey and ItemContainerDefaultKeyStyle. Each property returns a specialized object called a ResourceKey, which points to a style that you've defined in XAML. The DefaultStyleKey property points to the style that should be applied to configure the overall ListView. The ItemContainer.DefaultKeyStyle property points to the style that should be used to configure each ListViewItem in the ListView. Although these styles are free to tweak any property, they usually do

their work by replacing the ControlTemplate that's used for the ListView and the DataTemplate that's used for each ListViewItem.

Here's where the problems occur. The DataTemplate you use to display items is defined in XAML markup. Imagine you want to create a ListView that shows a tiled image for each item. This is easy enough using a DataTemplate—you simply need to bind the Source property of an Image to the correct property of your data object. But how do you know which data object the user will supply? If you hard-code property names as part of your view, you'll limit its usefulness, making it impossible to reuse your custom view in other scenarios. The alternative—forcing the user to supply the DataTemplate—means you can't pack as much functionality into the view, so reusing it won't be as useful.

■ **Tip** Before you begin creating a custom view, consider whether you could get the same result by simply using the right DataTemplate with a ListBox or a ListView/GridView combination.

So why go to all the effort of designing a custom view if you can already get all the functionality you need by restyling the ListView (or even the ListBox)? The primary reason is to create a list that can dynamically change views. For example, you might want a product list that can be viewed in different modes, depending on the user's selection. You could implement this by dynamically swapping in different DataTemplate objects (and this is a reasonable approach), but often a view needs to change both the DataTemplate of the ListViewItem and the layout or overall appearance of the ListView itself. A view helps clarify the relationship between these details in your source code.

The following example shows you how to create a grid that can be switched seamlessly from one view to another. The grid begins in the familiar column-separated view but also supports two tiled image views, as shown in Figure 22-3 and Figure 22-4.

Figure 22-3. *An image view*

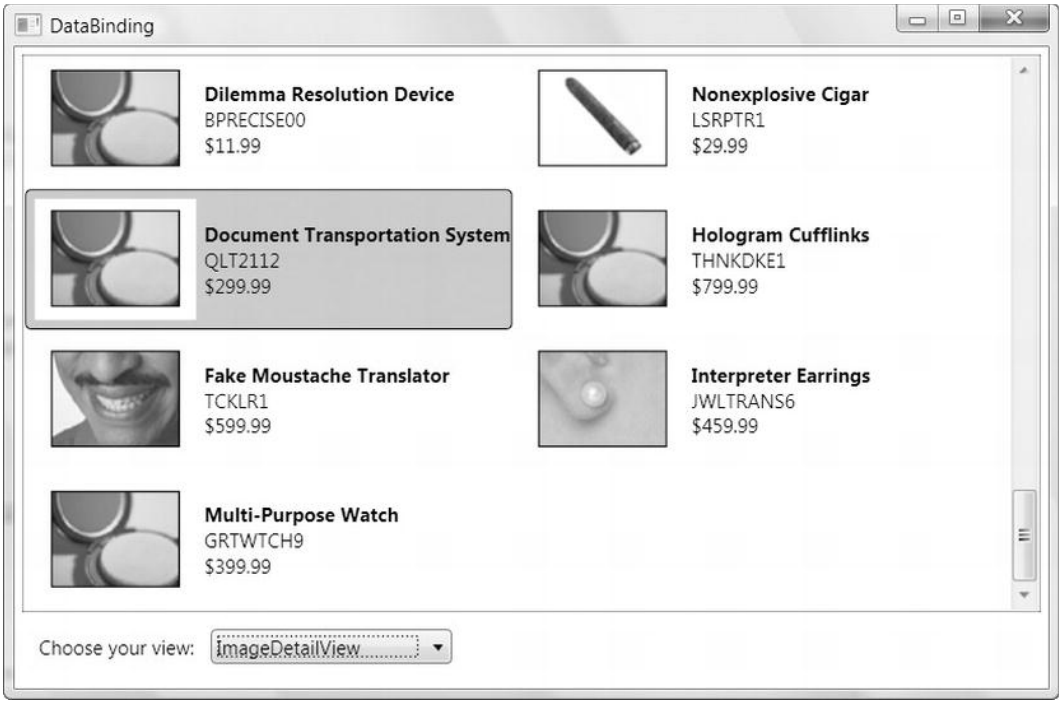

Figure 22-4. *A detailed image view*

The View Class

The first step that's required to build this example is to create the class that represents the custom view. This class must derive from ViewBase. In addition, it usually (although not always) overrides the DefaultStyleKey and ItemContainerDefaultStyleKey properties to supply style references.

In this example, the view is named TileView, because its key characteristic is that it tiles its items in the space provided. It uses a WrapPanel to lay out the contained ListViewItem objects. This view is not named ImageView, because the tile content isn't hard-coded and may not include images at all. Instead, the tile content is defined by using a template that the developer supplies when using the TileView.

The TileView class applies two styles: TileView (which applies to the ListView) and TileViewItem (which applies to the ListViewItem). Additionally, the TileView defines a property named ItemTemplate so the developer using the TileView can supply the correct data template. This template is then inserted inside each ListViewItem and used to create the tile content.

```
public class TileView : ViewBase
{
    private DataTemplate itemTemplate;
    public DataTemplate ItemTemplate
    {
        get { return itemTemplate; }
        set { itemTemplate = value; }
    }

    protected override object DefaultStyleKey
```

```
    {
        get { return new ComponentResourceKey(GetType(), "TileView"); }
    }

    protected override object ItemContainerDefaultStyleKey
    {
        get { return new ComponentResourceKey(GetType(), "TileViewItem"); }
    }
}
```

As you can see, the TileView class doesn't do much. It simply provides a ComponentResourceKey reference that points to the correct style. You first learned about the ComponentResourceKey in Chapter 10, when considering how you could retrieve shared resources from a DLL assembly.

The ComponentResourceKey wraps two pieces of information: the type of class that owns the style and a descriptive ResourceId string that identifies the resource. In this example, the type is obviously the TileView class for both resource keys. The descriptive ResourceId names aren't as important, but you'll need to be consistent. In this example, the default style key is named TileView, and the style key for each ListViewItem is named TileViewItem. In the following section, you'll dig into both these styles and see how they're defined.

The View Styles

For the TileView to work as written, WPF needs to be able to find the styles that you want to use. The trick to making sure styles are available automatically is creating a resource dictionary named generic.xaml. This resource dictionary must be placed in a project subfolder named Themes. WPF uses the generic.xaml file to get the default styles that are associated with a class. (You learned about this system when you considered custom control development in Chapter 18.)

In this example, the generic.xaml file defines the styles that are associated with the TileView class. To set up the association between your styles and the TileView, you need to give your style the correct key in the generic.xaml resource dictionary. Rather than using an ordinary string key, WPF expects your key to be a ComponentResourceKey object, and this ComponentResourceKey needs to match the information that's returned by the DefaultStyleKey and ItemContainerDefaultStyleKey properties of the TileView class.

Here's the basic structure of the generic.xaml resource dictionary, with the correct keys:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:DataBinding">

  <Style x:Key="{ComponentResourceKey TypeInTargetAssembly={x:Type local:TileView},
ResourceId=TileView}"
   TargetType="{x:Type ListView}"
   BasedOn="{StaticResource {x:Type ListBox}}">
    ...
  </Style>

  <Style x:Key="{ComponentResourceKey TypeInTargetAssembly={x:Type local:TileView},
ResourceId=TileViewItem}"
   TargetType="{x:Type ListViewItem}"
   BasedOn="{StaticResource {x:Type ListBoxItem}}">
    ...
```

```
  </Style>

</ResourceDictionary>
```

As you can see, the key of each style is set to match the information provided by the TileView class. Additionally, the styles also set the TargetType property (to indicate which element the style modifies) and the BasedOn property (to inherit basic style settings from more fundamental styles used with the ListBox and ListBoxItem). This saves some work, and it allows you to focus on extending these styles with custom settings.

Because these two styles are associated with the TileView, they'll be used to configure the ListView whenever you've set the View property to a TileView object. If you're using a different view object, these styles will be ignored. This is the magic that makes the ListView work the way you want, so that it seamlessly reconfigures itself every time you change the View property.

The TileView style that applies to the ListView makes three changes:

- It adds a slightly different border around the ListView.
- It sets the attached Grid.IsSharedSizeScope property to true. This allows different list items to use shared column or row settings if they use the Grid layout container (a feature first explained in Chapter 3). In this example, it makes sure each item has the same dimensions in the detailed tile view.
- It changes the ItemsPanel from a StackPanel to a WrapPanel, allowing the tiling behavior. The WrapPanel width is set to match the width of the ListView.

Here's the full markup for this style:

```
<Style x:Key="{ComponentResourceKey TypeInTargetAssembly={x:Type local:TileView},
 ResourceId=TileView}"
 TargetType="{x:Type ListView}" BasedOn="{StaticResource {x:Type ListBox}}">
  <Setter Property="BorderBrush" Value="Black"></Setter>
  <Setter Property="BorderThickness" Value="0.5"></Setter>
  <Setter Property="Grid.IsSharedSizeScope" Value="True"></Setter>

  <Setter Property="ItemsPanel">
    <Setter.Value>
      <ItemsPanelTemplate>
        <WrapPanel Width="{Binding (FrameworkElement.ActualWidth),
                           RelativeSource={RelativeSource
                           AncestorType=ScrollContentPresenter}}">
        </WrapPanel>
      </ItemsPanelTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

These are relatively minor changes. A more ambitious view could link to a style that changes the control template that's used for the ListView, modifying it much more dramatically. This is where you begin to see the benefits of the view model. By changing a single property in the ListView, you can apply a combination of related settings through two styles. The TileView style that applies to the ListViewItem changes a few other details. It sets the padding and content alignment and, most important, sets the DataTemplate that's used to display content.

Here's the full markup for this style:

```
<Style x:Key="{ComponentResourceKey TypeInTargetAssembly={x:Type local:TileView},
 ResourceId=TileViewItem}"
 TargetType="{x:Type ListViewItem}"
 BasedOn="{StaticResource {x:Type ListBoxItem}}">
  <Setter Property="Padding" Value="3"/>
  <Setter Property="HorizontalContentAlignment" Value="Center"></Setter>
  <Setter Property="ContentTemplate" Value="{Binding Path=View.ItemTemplate,
   RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type ListView}
                }}"></Setter>
</Style>
```

Remember that to ensure maximum flexibility, the TileView is designed to use a data template that's supplied by the developer. To apply this template, the TileView style needs to retrieve the TileView object (using the ListView.View property) and then pull the data template from the TileView.ItemTemplate property. This step is performed using a binding expression that searches up the element tree (using the FindAncestor RelativeSource mode) until it finds the containing ListView.

■ **Note** Rather than setting the ListViewItem.ContentTemplate property, you could achieve the same result by setting the ListView.ItemTemplate property. It's really just a matter of preference.

Using the ListView

After you've built your view class and the supporting styles, you're ready to put them to use in a ListView control. To use a custom view, you simply need to set the ListView.View property to an instance of your view object, as shown here:

```
<ListView Name="lstProducts">
  <ListView.View>
    <TileView ... >
  </ListView.View>
</ListView>
```

However, this example demonstrates a ListView that can switch between three views. As a result, you need to instantiate three distinct view objects. The easiest way to manage this is to define each view object separately in the Windows.Resources collection. You can then load the view you want when the user makes a selection from the ComboBox control, by using this code:

```
private void lstView_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    ComboBoxItem selectedItem = (ComboBoxItem)lstView.SelectedItem;
    lstProducts.View = (ViewBase)this.FindResource(selectedItem.Content);
}
```

The first view is simple enough—it uses the familiar GridView class that you considered earlier to create a multicolumn display. Here's the markup it uses:

```
<GridView x:Key="GridView">
  <GridView.Columns>
    <GridViewColumn Header="Name"
     DisplayMemberBinding="{Binding Path=ModelName}" />
```

```
    <GridViewColumn Header="Model"
     DisplayMemberBinding="{Binding Path=ModelNumber}" />
    <GridViewColumn Header="Price"
      DisplayMemberBinding="{Binding Path=UnitCost, StringFormat={}{0:C}}" />
  </GridView.Columns>
</GridView>
```

The two TileView objects are more interesting. Both of them supply a template to determine what the tile looks like. The ImageView (shown in Figure 22-3) uses a StackPanel that stacks the product image above the product title:

```
<local:TileView x:Key="ImageView">
  <local:TileView.ItemTemplate>
    <DataTemplate>
      <StackPanel Width="150" VerticalAlignment="Top">
        <Image Source="{Binding Path=ProductImagePath,
                        Converter={StaticResource ImagePathConverter}}">
        </Image>
        <TextBlock TextWrapping="Wrap" HorizontalAlignment="Center"
         Text="{Binding Path=ModelName}"></TextBlock>
      </StackPanel>
    </DataTemplate>
  </local:TileView.ItemTemplate>
</local:TileView>
```

The ImageDetailView uses a two-column grid. A small version of the image is placed on the left, and more-detailed information is placed on the right. The second column is placed into a shared-size group so that all the items have the same width (as determined by the largest text value).

```
<local:TileView x:Key="ImageDetailView">
  <local:TileView.ItemTemplate>
    <DataTemplate>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="Auto"></ColumnDefinition>
          <ColumnDefinition Width="Auto" SharedSizeGroup="Col2"></ColumnDefinition>
        </Grid.ColumnDefinitions>

        <Image Margin="5"  Width="100"
         Source="{Binding Path=ProductImagePath,
                  Converter={StaticResource ImagePathConverter}}">
        </Image>
        <StackPanel Grid.Column="1" VerticalAlignment="Center">
          <TextBlock FontWeight="Bold" Text="{Binding Path=ModelName}"></TextBlock>
          <TextBlock Text="{Binding Path=ModelNumber}"></TextBlock>
          <TextBlock Text="{Binding Path=UnitCost, StringFormat={}{0:C}}">
          </TextBlock>
        </StackPanel>
      </Grid>
    </DataTemplate>
  </local:TileView.ItemTemplate>
</local:TileView>
```

This is undoubtedly more code than you wanted to generate to create a ListView with multiple viewing options. However, the example is now complete, and you can easily create additional views (based on the TileView class) that supply different item templates and give you even more viewing options.

Passing Information to a View

You can make your view classes more flexible by adding properties that the consumer can set when using the view. Your style can then retrieve these values by using data binding and apply them to configure the Setter objects.

For example, the TileView currently highlights selected items with an unattractive blue color. The effect is all the more jarring because it makes the black text with the product details more difficult to read. As you probably remember from Chapter 17, you can fix these details by using a customized control template with the correct triggers.

But rather than hard-code a set of pleasing colors, it makes sense to let the view consumer specify this detail. To do this with the TileView, you could add a set of properties like these:

```
private Brush selectedBackground = Brushes.Transparent;
public Brush SelectedBackground
{
    get { return selectedBackground; }
    set { selectedBackground = value; }
}

private Brush selectedBorderBrush = Brushes.Black;
public Brush SelectedBorderBrush
{
    get { return selectedBorderBrush; }
    set { selectedBorderBrush = value; }
}
```

Now you can set these details when instantiating a view object:

```
<local:TileView x:Key="ImageDetailView" SelectedBackground="LightSteelBlue">
  ...
</local:TileView>
```

The final step is to use these colors in the ListViewItem style. To do so, you need to add a Setter that replaces the ControlTemplate. In this case, a simple rounded border is used with a ContentPresenter. When the item is selected, a trigger fires and applies the new border and background colors:

```
<Style x:Key="{ComponentResourceKey TypeInTargetAssembly={x:Type local:TileView},
 ResourceId=TileViewItem}"
 TargetType="{x:Type ListViewItem}"
 BasedOn="{StaticResource {x:Type ListBoxItem}}">
  ...
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBoxItem}">
        <Border Name="Border" BorderThickness="1" CornerRadius="3">
          <ContentPresenter />
        </Border>
        <ControlTemplate.Triggers>
```

```
          <Trigger Property="IsSelected" Value="True">
            <Setter TargetName="Border" Property="BorderBrush"
             Value="{Binding Path=View.SelectedBorderBrush,
                      RelativeSource={RelativeSource Mode=FindAncestor,
                      AncestorType={x:Type ListView}}}"></Setter>
            <Setter TargetName="Border" Property="Background"
            Value="{Binding Path=View.SelectedBackground,
                    RelativeSource={RelativeSource Mode=FindAncestor,
                    AncestorType={x:Type ListView}}}"></Setter>
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Figure 22-3 and Figure 22-4 show this selection behavior. Figure 22-3 uses a transparent background, and Figure 22-4 uses a light blue highlight color.

■ **Note** Unfortunately, this technique of passing information to a view still doesn't help you make a truly generic view. That's because there's no way to modify the data templates based on this information.

The TreeView

The TreeView is a Windows staple, and it's a common ingredient in everything from the Windows Explorer file browser to the .NET help library. WPF's implementation of the TreeView is impressive, because it has full support for data binding.

The TreeView is, at its heart, a specialized ItemsControl that hosts TreeViewItem objects. But unlike the ListViewItem, the TreeViewItem is not a content control. Instead, each TreeViewItem is a separate ItemsControl, with the ability to hold more TreeViewItem objects. This flexibility allows you to create a deeply layered data display.

■ **Note** Technically, the TreeViewItem derives from HeaderedItemsControl, which derives from ItemsControl. The HeaderedItemsControl class adds a Header property, which holds the content (usually text) that you want to display for that item in the tree. WPF includes two other HeaderedItemsControl classes: the MenuItem and the ToolBar.

Here's the skeleton of a very basic TreeView, which is declared entirely in markup:

```
<TreeView>
  <TreeViewItem Header="Fruit">
    <TreeViewItem Header="Orange"/>
    <TreeViewItem Header="Banana"/>
    <TreeViewItem Header="Grapefruit"/>
  </TreeViewItem>
 <TreeViewItem Header="Vegetables">
```

```
    <TreeViewItem Header="Aubergine"/>
    <TreeViewItem Header="Squash"/>
    <TreeViewItem Header="Spinach"/>
  </TreeViewItem>
</TreeView>
```

It's not necessary to construct a TreeView out of TreeViewItem objects. In fact, you have the ability to add virtually any element to a TreeView, including buttons, panels, and images. However, if you want to display nontext content, the best approach is to use a TreeViewItem wrapper and supply your content through the TreeViewItem.Header property. This gives you the same effect as adding non-TreeViewItem elements directly to your TreeView but makes it easier to manage a few TreeView-specific details, such as selection and node expansion. If you want to display a non-UIElement object, you can format it by using data templates with the HeaderTemplate or HeaderTemplateSelector property.

Creating a Data-Bound TreeView

Usually, you won't fill a TreeView with fixed information that's hard-coded in your markup. Instead, you'll construct the TreeViewItem objects you need programmatically, or you'll use data binding to display a collection of objects.

Filling a TreeView with data is easy enough—as with any ItemsControl, you simply set the ItemsSource property. However, this technique fills only the first level of the TreeView. A more interesting use of the TreeView incorporates *hierarchical data* that has some sort of nested structure.

For example, consider the TreeView shown in Figure 22-5. The first level consists of Category objects, and the second level shows the Product objects that fall into each category.

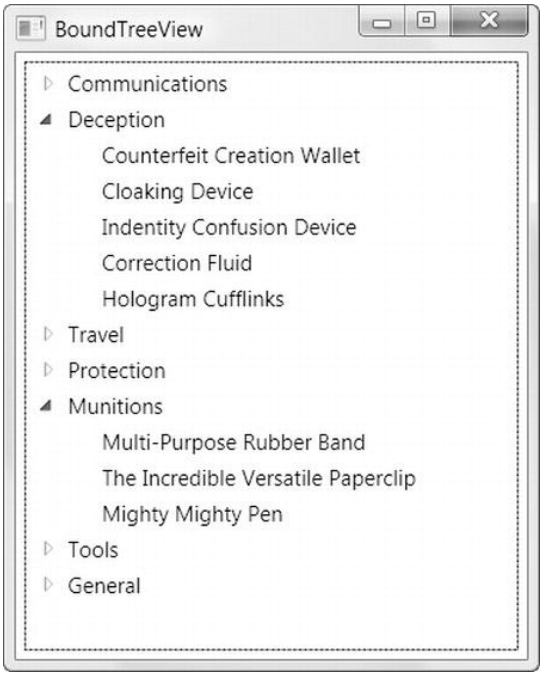

Figure 22-5. *A TreeView of categories and products*

The TreeView makes hierarchical data display easy, whether you're working with handcrafted classes or the ADO.NET DataSet. You simply need to specify the correct data templates. Your templates indicate the relationship between the different levels of the data.

For example, imagine you want to build the example shown in Figure 22-5. You've already seen the Products class that's used to represent a single Product. But to create this example, you also need a Category class. Like the Product class, the Category class implements INotifyPropertyChanged to provide change notifications. The only new detail is that the Category class exposes a collection of Product objects through its Product property.

```
public class Category : INotifyPropertyChanged
{
    private string categoryName;
    public string CategoryName
    {
        get { return categoryName; }
        set { categoryName = value;
              OnPropertyChanged(new PropertyChangedEventArgs("CategoryName"));
            }
    }

    private ObservableCollection<Product> products;
    public ObservableCollection<Product> Products
    {
        get { return products; }
        set { products = value;
              OnPropertyChanged(new PropertyChangedEventArgs("Products"));
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, e);
    }

    public Category(string categoryName, ObservableCollection<Product> products)
    {
        CategoryName = categoryName;
        Products = products;
    }
}
```

■ **Tip** This trick—creating a collection that exposes another collection through a property—is the secret to navigating parent-child relationships with WPF data binding. For example, you can bind a collection of Category objects to one list control, and then bind another list control to the Products property of the currently selected Category object to show the related Product objects.

To use the Category class, you also need to modify the data access code that you first saw in Chapter 19. Now you'll query the information about products and categories from the database. In this example, the window calls the StoreDB.GetCategoriesAndProducts() method to get a collection of Category objects, each of which has a nested collection of Product objects. The Category collection is then bound to the tree so that it will appear in the first level:

```
treeCategories.ItemsSource = App.StoreDB.GetCategoriesAndProducts();
```

To display the categories, you need to supply a TreeView.ItemTemplate that can process the bound objects. In this example, you need to display the CategoryName property of each Category object. Here's the data template that does it:

```
<TreeView Name="treeCategories" Margin="5">
  <TreeView.ItemTemplate>
    <HierarchicalDataTemplate>
      <TextBlock Text="{Binding Path=CategoryName}" />
    </HierarchicalDataTemplate>
  </TreeView.ItemTemplate>
</TreeView>
```

The only unusual detail here is that the TreeView.ItemTemplate is set by using a HierarchicalDataTemplate object instead of a DataTemplate. The HierarchicalDataTemplate has the added advantage that it can wrap a second template. The HierarchicalDataTemplate can then pull a collection of items from the first level and provide that to the second-level template. You simply set the ItemsSource property to identify the property that has the child items, and you set the ItemTemplate property to indicate how each object should be formatted.

Here's the revised data template:

```
<TreeView Name="treeCategories" Margin="5">
  <TreeView.ItemTemplate>
    <HierarchicalDataTemplate ItemsSource="{Binding Path=Products}">
      <TextBlock Text="{Binding Path=CategoryName}" />
      <HierarchicalDataTemplate.ItemTemplate>
        <DataTemplate>
          <TextBlock Text="{Binding Path=ModelName}" />
        </DataTemplate>
      </HierarchicalDataTemplate.ItemTemplate>
    </HierarchicalDataTemplate>
  </TreeView.ItemTemplate>
</TreeView>
```

Essentially, you now have two templates, one for each level of the tree. The second template uses the selected item from the first template as its data source.

Although this markup works perfectly well, it's common to factor out each data template and apply it to your data objects by data type instead of by position. To understand what that means, it helps to consider a revised version of the markup for the data-bound TreeView:

```
<Window x:Class="DataBinding.BoundTreeView" ...
    xmlns:local="clr-namespace:DataBinding">
  <Window.Resources>
    <HierarchicalDataTemplate DataType="{x:Type local:Category}"
     ItemsSource="{Binding Path=Products}">
      <TextBlock Text="{Binding Path=CategoryName}"/>
    </HierarchicalDataTemplate>
```

```
    <HierarchicalDataTemplate DataType="{x:Type local:Product}">
      <TextBlock Text="{Binding Path=ModelName}" />
    </HierarchicalDataTemplate>
  </Window.Resources>

  <Grid>
    <TreeView Name="treeCategories" Margin="5">
    </TreeView>
  </Grid>
</Window>
```

In this example, the TreeView doesn't explicitly set its ItemTemplate. Instead, the appropriate ItemTemplate is used based on the data type of the bound object. Similarly, the Category template doesn't specify the ItemTemplate that should be used to process the Products collection. It's also chosen automatically by data type. This tree is now able to show a list of products or a list of categories that contain groups of products.

In the current example, these changes don't add anything new. This approach simplifies the markup and makes it easier to reuse your templates, but it doesn't affect the way your data is displayed. However, if you have deeply nested trees that have looser structures, this design is invaluable. For example, imagine you're creating a tree of Manager objects, and each Manager object has an Employees collection. This collection might contain ordinary Employee objects or other Manager objects, which would in turn contain more Employees. If you use the type-based template system shown earlier, each object automatically gets the template that's right for its data type.

Binding a DataSet to a TreeView

You can also use a TreeView to show a multilayered DataSet—one that has relationships linking one DataTable to another.

For example, here's a code routine that creates a DataSet, fills it with a table of products and a separate table of categories, and links the two tables together with a DataRelation object:

```
public DataSet GetCategoriesAndProductsDataSet()
{
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand("GetProducts", con);
    cmd.CommandType = CommandType.StoredProcedure;
    SqlDataAdapter adapter = new SqlDataAdapter(cmd);

    DataSet ds = new DataSet();
    adapter.Fill(ds, "Products");
    cmd.CommandText = "GetCategories";
    adapter.Fill(ds, "Categories");

    // Set up a relation between these tables.
    DataRelation relCategoryProduct = new DataRelation("CategoryProduct",
      ds.Tables["Categories"].Columns["CategoryID"],
      ds.Tables["Products"].Columns["CategoryID"]);
      ds.Relations.Add(relCategoryProduct);

    return ds;
```

```
}
```

To use this in a TreeView, you begin by binding to the DataTable you want to use for the first level:

```
DataSet ds = App.StoreDB.GetCategoriesAndProductsDataSet();
treeCategories.ItemsSource = ds.Tables["Categories"].DefaultView;
```

But how do you get the related rows? After all, you can't call a method such as GetChildRows() from XAML. Fortunately, the WPF data-binding system has built-in support for this scenario. The trick is to use the name of your DataRelation as the ItemsSource for your second level. In this example, the DataRelation was created with the name CategoryProduct, so this markup does the trick:

```
<TreeView Name="treeCategories" Margin="5">
  <TreeView.ItemTemplate>
    <HierarchicalDataTemplate ItemsSource="{Binding CategoryProduct}">
      <TextBlock Text="{Binding CategoryName}" Padding="2" />
        <HierarchicalDataTemplate.ItemTemplate>
          <DataTemplate>
            <TextBlock Text="{Binding ModelName}" Padding="2" />
          </DataTemplate>
        </HierarchicalDataTemplate.ItemTemplate>
      </HierarchicalDataTemplate>
    </TreeView.ItemTemplate>
</TreeView>
```

Now this example works in the same way as the previous example, which used custom Product and Category objects.

Just-in-Time Node Creation

TreeView controls are often used to hold huge amounts of data. That's because the TreeView display is collapsible. Even if the user scrolls from top to bottom, not all the information is necessarily visible. The information that isn't visible can be omitted from the TreeView altogether, reducing its overhead (and the amount of time required to fill the tree). Even better, each TreeViewItem fires an Expanded event when it's opened and a Collapsed event when it's closed. You can use this point in time to fill in missing nodes or discard ones that you don't need. This technique is called *just-in-time node creation*.

Just-in-time node creation can be applied to applications that pull their data from a database, but the classic example is a directory-browsing application. In current times, most people have huge, sprawling hard drives. Although you could fill a TreeView with the directory structure of a hard drive, the process is aggravatingly slow. A better idea is to begin with a partially collapsed view and allow the user to dig down into specific directories (as shown in Figure 22-6). As each node is opened, the corresponding subdirectories are added to the tree—a process that's nearly instantaneous.

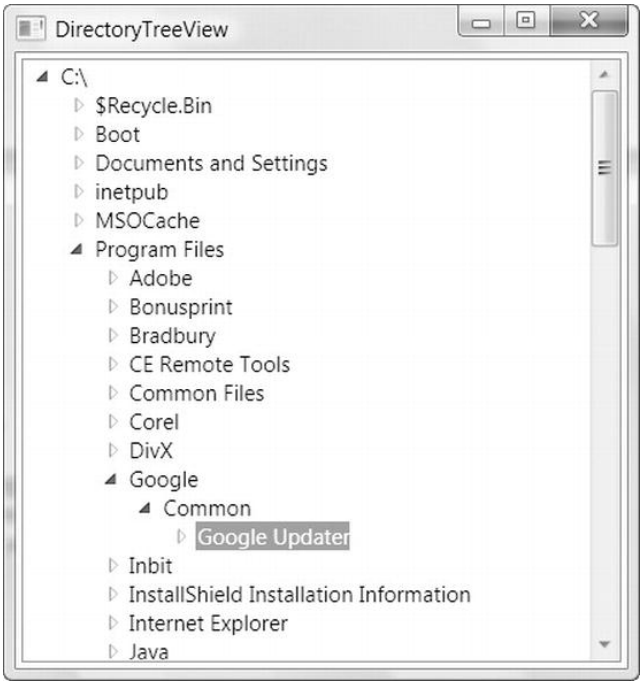

Figure 22-6. *Digging into a directory tree*

Using a just-in-time TreeView to display the folders on a hard drive is nothing new. (In fact, the technique is demonstrated in my book *Pro .NET 2.0 Windows Forms and Custom Controls in C#* [Apress, 2005].) However, event routing makes the WPF solution just a bit more elegant.

The first step is to add a list of drives to the TreeView when the window first loads. Initially, the node for each drive is collapsed. The drive letter is displayed in the header, and the DriveInfo object is stored in the TreeViewItem.Tag property to make it easier to find the nested directories later without re-creating the object. (This increases the memory overhead of the application, but it also reduces the number of file-access security checks. The overall effect is small, but it improves performance slightly and simplifies the code.)

Here's the code that fills the TreeView with a list of drives, using the System.IO.DriveInfo class:

```
foreach (DriveInfo drive in DriveInfo.GetDrives())
{
    TreeViewItem item = new TreeViewItem();
    item.Tag = drive;
    item.Header = drive.ToString();

    item.Items.Add("*");
    treeFileSystem.Items.Add(item);
}
```

This code adds a placeholder (a string with an asterisk) under each drive node. The placeholder is not shown, because the node begins in a collapsed state. As soon as the node is expanded, you can remove the placeholder and add the list of subdirectories in its place.

■ **Note** The placeholder is a useful tool that can allow you to determine whether the user has expanded this folder to view its contents yet. However, the primary purpose of the placeholder is to make sure the expand icon appears next to this item. Without that, the user won't be able to expand the directory to look for subfolders. If the directory doesn't include any subfolders, the expand icon will simply disappear when the user attempts to expand it, which is similar to the behavior of Windows Explorer when viewing network folders.

To perform the just-in-time node creation, you must handle the TreeViewItem.Expanded event. Because this event uses bubbling, you can attach an event handler directly on the TreeView to handle the Expanded event for any TreeViewItem inside:

```
<TreeView Name="treeFileSystem" TreeViewItem.Expanded="item_Expanded">
</TreeView>
```

Here's the code that handles the event and fills in the missing next level of the tree by using the System.IO.DirectoryInfo class:

```
private void item_Expanded(object sender, RoutedEventArgs e)
{
    TreeViewItem item = (TreeViewItem)e.OriginalSource;
    item.Items.Clear();

    DirectoryInfo dir;
    if (item.Tag is DriveInfo)
    {
        DriveInfo drive = (DriveInfo)item.Tag;
        dir = drive.RootDirectory;
    }
    else
    {
        dir = (DirectoryInfo)item.Tag;
    }

    try
    {
        foreach (DirectoryInfo subDir in dir.GetDirectories())
        {
            TreeViewItem newItem = new TreeViewItem();
            newItem.Tag = subDir;
            newItem.Header = subDir.ToString();
            newItem.Items.Add("*");
            item.Items.Add(newItem);
        }
    }
    catch
    {
        // An exception could be thrown in this code if you don't
        // have sufficient security permissions for a file or directory.
        // You can catch and then ignore this exception.
    }
```

```
}
```

Currently, this code performs a refresh every time the item is expanded. Optionally, you could perform this only the first time it's expanded, when the placeholder is found. This reduces the work your application needs to do, but it increases the chance of out-of-date information. Alternatively, you could perform a refresh every time an item is selected by handling the TreeViewItem.Selected event, or you could use a component such as the System.IO.FileSystemWatcher to wait for operating system notifications when a folder is added, removed, or renamed. The FileSystemWatcher is the only way to ensure that you update the directory tree immediately when a change happens, but it also has the greatest overhead.

CREATING ADVANCED TREEVIEW CONTROLS

There's a lot that you can accomplish when you combine the power of control templates (discussed in Chapter 17) with the TreeView. In fact, you can create a control that looks and behaves in a radically different way simply by replacing the templates for the TreeView and TreeViewItem controls.

Making these adjustments requires some deeper template exploration. You can get started with some eye-opening examples. Visual Studio includes a sample of a multicolumned TreeView that unites a tree with a grid. To browse it, look for the index entry *TreeListView Sample [WPF]* in the Visual Studio help. Another intriguing example is Josh Smith's layout experiment, which transforms the TreeView into something that more closely resembles an organizational chart. You can view the full code at `www.codeproject.com/KB/WPF/CustomTreeViewLayout.aspx`.

The DataGrid

As its name suggests, the DataGrid is a data-display control that takes the information from a collection of objects and renders it in a grid of rows and cells. Each row corresponds to a separate object, and each column corresponds to a property in that object.

The DataGrid adds much-needed versatility for dealing with data in WPF. Its column-based model gives it remarkable formatting flexibility. Its selection model allows you to choose whether users can select a row, multiple rows, or some combination of cells. Its editing support is powerful enough that you can use the DataGrid as an all-in-one data editor for simple and complex data.

To create a quick-and-dirty DataGrid, you can use automatic column generation. To do so, you need to set the AutoGenerateColumns property to true (which is the default value):

```
<DataGrid x:Name="gridProducts" AutoGenerateColumns="True">
</DataGrid>
```

Now you can fill the DataGrid as you fill a list control, by setting the ItemsSource property:

```
gridProducts.DataSource = products;
```

Figure 22-7 shows a DataGrid that uses automatic column generation with the collection of Product objects. For automatic column generation, the DataGrid uses reflection to find every public property in the bound data object. It creates a column for each property.

Figure 22-7. *A DataGrid with automatically generated columns*

To display nonstring properties, the DataGrid calls ToString(), which works well for numbers, dates, and other simple data types, but it won't work as well if your objects include a more complex data object. (In this case, you may want to explicitly define your columns, which gives you the chance to bind to a subproperty, use a value converter, or apply a template to get the correct display content.)

Table 22-1 lists some of the properties you can use to customize a DataGrid's basic appearance. In the following sections, you'll see how to get fine-grained formatting control with styles and templates. You'll also see how the DataGrid deals with sorting and selection, and you'll consider many more properties that underlie these features.

Table 22-1. *Basic Display Properties for the DataGrid*

Name	Description
RowBackground and AlternatingRowBackground	The brush that's used to paint the background behind every row (RowBackground) and whether alternate rows are painted with a different background color (AlternatingRowBackground), making it easier to distinguish rows at a glance. By default, the DataGrid gives odd-numbered rows a white background and even-numbered rows a light-gray background.
ColumnHeaderHeight	The height (in device-independent units) of the row that has the column headers at the top of the DataGrid.
RowHeaderWidth	The width (in device-independent units) of the column that has the row headers. This is the column at the far left of the grid, which does not show any data. It indicates the currently selected row (with an arrow) and when the row is being edited (with an arrow in a circle).

Name	Description
ColumnWidth	The sizing mode that's used to set the default width of every column, as a DataGridLength object. (The following section explains your column-sizing options.)
RowHeight	The height of every row. This setting is useful if you plan to display multiple lines of text or different content (for example, images) in the DataGrid. Unlike columns, rows cannot be resized by the user.
GridLinesVisibility	A value from the DataGridGridlines enumeration that determines which grid lines are shown (Horizontal, Vertical, None, or All).
VerticalGridLinesBrush	The brush that's used to paint the grid lines between columns.
HorizontalGridLinesBrush	The brush that's used to paint the grid lines between rows.
HeadersVisibility	A value from the DataGridHeaders enumeration that determines which headers are shown (Column, Row, All, None).
HorizontalScrollBarVisibility and VerticalScrollBarVisibility	A value from the ScrollBarVisibility enumeration that determines whether a scrollbar is shown when needed (Auto), always (Visible), or never (Hidden). The default for both properties is Auto.

Resizing and Rearranging Columns

When displaying automatically generated columns, the DataGrid attempts to size the width of each column intelligently, according to the DataGrid.ColumnWidth property.

To set the ColumnWidth property, you supply a DataGridLength object. Your DataGridLength can specify an exact size (in device-independent units) or a special sizing mode, which gets the DataGrid to do some of the work for you. If you choose to use an exact size, simply set ColumnWidth equal to the appropriate number (in XAML) or supply the number as a constructor argument when creating the DataGridLength (in code):

```
grid.ColumnWidth = new DataGridLength(150);
```

The specialized sizing modes are more interesting. You access them through the static properties of the DataGridLength class. Here's an example that uses the default DataGridLength.SizeToHeader sizing mode, which means the columns are made wide enough to accommodate their header text:

```
grid.ColumnWidth = DataGridLength.SizeToHeader;
```

Another popular option is DataGridLength.SizeToCells, which widens each column to fit the widest value that's currently in view. The DataGrid attempts to preserve this intelligent sizing approach when the user starts scrolling through the data. As soon as you come across a row with longer data, the DataGrid widens the appropriate columns to fit it. This automatic sizing is one-way only, so columns don't shrink when you leave large data behind.

Your other special sizing mode choice is DataGridLength.Auto, which works just like DataGridLength.SizeToCells, except that each column is widened to fit the largest displayed value *or* the column header text—whichever is wider.

The DataGrid also allows you to use a proportional sizing system that parallels the star-sizing in the Grid layout panel. Once again, * represents proportional sizing, and you can add a number to split the available space by using the ratios you pick (say, 2* and * to give the first column twice the space of the second). To set up this sort of relationship, or to give your columns different widths or sizing modes, you

need to explicitly set the Width property of each column object. You'll see how to explicitly define and configure DataGrid columns in the next section.

The automatic sizing of the DataGrid columns is interesting and often useful, but it's not always what you want. Consider the example shown in Figure 22-7, which contains a Description column that holds a long string of text. Initially, the Description column is made extremely wide to fit this data, crowding the other columns out of the way. (In Figure 22-7, the user has manually resized the Description column to a more sensible size. All the other columns are left at their initial widths.) After a column has been resized, it doesn't exhibit the automatic enlarging behavior when the user scrolls through the data.

■ **Tip** Of course, you don't want to force your users to grapple with ridiculously wide columns. For that reason, you'll also choose to define a different column width or different sizing mode for each column. To do this, you need to define your columns explicitly and set the DataGridColumn.Width property. When set on a column, this property overrides the DataGrid.ColumnWidth default. You'll learn how to define your columns explicitly in the next section.

Ordinarily, users can resize columns by dragging the column edge to either size. You can prevent the user from resizing the columns in your DataGrid by setting the CanUserResizeColumns property to false. If you want to be more specific, you can prevent the user from resizing an individual column by setting the CanUserResize property of that column to false. You can also prevent the user from making the column extremely narrow by setting the column's MinWidth property.

The DataGrid has another surprise frill that lets users customize the column display. Not only can columns be resized, but they can also be dragged from one position to another. If you don't want users to have this reordering ability, set the CanUserReorderColumns property of the DataGrid or the CanUserReorder property of a specific column to false.

Defining Columns

Using automatically generated columns, you can quickly create a DataGrid that shows all your data. However, you give up a fair bit of control. For example, you can't control how columns are ordered, how wide they are, how the values inside are formatted, and what header text is placed at the top.

A far more powerful approach is to turn off automatic column generation by setting AutoGenerateColumns to false. You can then define the columns you want explicitly, with the settings you want and in the order you specify. To do this, you need to fill the DataGrid.Columns collection with the correct column objects.

Currently, the DataGrid supports several types of columns, which are represented by different classes that derive from DataGridColumn:

DataGridTextColumn: This column is the standard choice for most data types. The value is converted to text and displayed in a TextBlock. When you edit the row, the TextBlock is replaced with a standard text box.

DataGridCheckBoxColumn: This column shows a check box. This column type is used automatically for Boolean (or nullable Boolean) values. Ordinarily, the check box is read-only, but when you edit the row, it becomes a normal check box.

DataGridHyperlinkColumn: This column shows a clickable link. If used in conjunction with WPF navigation containers such as the Frame or NavigationWindow, it allows the user to navigate to another URI (typically, an external website).

DataGridComboBox: This column looks like a DataGridTextColumn initially, but changes to a drop-down ComboBox in edit mode. It's a good choice when you want to constrain edits to a small set of allowed values.

DataGridTemplateColumn: This column is by far the most powerful option. It allows you to define a data template for displaying column values, with all the flexibility and power you have when using templates in a list control. For example, you can use a DataGridTemplateColumn to display image data or to use a specialized WPF control (such as a drop-down list with valid values or a DatePicker for date values).

For example, here's a revised DataGrid that creates a two-column display with product names and prices. It also applies clearer column captions and widens the Product column to fit its data:

```
<DataGrid x:Name="gridProducts" Margin="5" AutoGenerateColumns="False">
  <DataGrid.Columns>
    <DataGridTextColumn Header="Product" Width="175"
     Binding="{Binding Path=ModelName}"></DataGridTextColumn>
    <DataGridTextColumn Header="Price"
     Binding="{Binding Path=UnitCost}"></DataGridTextColumn>
  </DataGrid.Columns>
</DataGrid>
```

When you define a column, you almost always set three details: the header text that appears at the top of the column, the width of the column, and the binding that gets the data.

Usually, you'll use a simple string to set the DataGridColumn.Header property, but there's no need to stick to ordinary text. The column header acts as content control, and you can supply any element for the Header property, including an image or a layout panel with a combination of elements.

The DataGridColumn.Width property supports hard-coded values and several automatic sizing modes, just like the DataGrid.ColumnWidth property you considered in the previous section. The only difference is that DataGridColumn.Width applies to a single column, while DataGrid.ColumnWidth sets the default for the whole table. When DataGridColumn.Width is set, it overrides the DataGrid.ColumnWidth.

The most important detail is the binding expression that provides the correct information for the column, as set by the DataGridColumn.Binding property. This approach is different from the simple list controls such as the ListBox and ComboBox. These controls include a DisplayMemberPath property instead of a Binding property. The Binding approach is more flexible—it allows you to use string formatting and value converters without needing to switch to a full-fledged template column.

```
<DataGridTextColumn Header="Price" Binding=
 "{Binding Path=UnitCost, StringFormat={}{0:C}}">
</DataGridTextColumn>
```

■ **Tip** You can dynamically show and hide columns by modifying the Visibility property of the corresponding column object. Additionally, you can move columns at any time by changing their DisplayIndex values.

DataGridCheckBoxColumn

The Product class doesn't include any Boolean properties. If it did, the DataGridCheckBoxColumn would be a useful option.

As with DataGridTextColumn, the Binding property extracts the data—in this case, the true or false value that's used to set the IsChecked property of the CheckBox element inside. The DataGridCheckBoxColumn also adds a property named Content that lets you show optional content alongside the check box. Finally, the DataGridCheckBoxColumn includes an IsThreeState property, which determines whether the check box supports the undetermined state as well as the more obvious checked and unchecked states. If you're using the DataGridCheckBoxColumn to show the information from a nullable Boolean value, you can set IsThreeState property to true. That way, the user can click back to the undetermined state (which shows a lightly shaded check box) to return the bound value to null.

DataGridHyperlinkColumn

The DataGridHyperlinkColumn allows you to display text values that contain a single URL each. For example, if the Product class has a string property named ProductLink, and that property contained values such as *http://myproducts.com/info?productID=10432*, you could display this information in a DataGridHyperlinkColumn. Every bound value would be displayed using the Hyperlink element, and rendered like this:

```
<Hyperlink NavigateUri="http://myproducts.com/info?productID=10432"
 >http://myproducts.com/info?productID=10432</Hyperlink>
```

Then the user could click a hyperlink to trigger navigation and visit the related page, with no code required. However, there's a major caveat: this automatic navigation trick works only if you've placed your DataGrid in a container that supports navigation events, such as the Frame or NavigationWindow. You'll learn about both controls and the Hyperlink in Chapter 24. If you want a more versatile way to accomplish a similar effect, consider using the DataGridTemplateColumn. You can use it to show underlined, clickable text (in fact, you can even use the Hyperlink control), but you'll have the flexibility of handling click events in your code.

Ordinarily, the DataGridHyperlinkColumn uses the same piece of information for navigation and for display. However, you can specify these details separately if you want. To do so, just set the URI with the Binding property, and use the optional ContentBinding property to get display text from a different property in the bound data object.

DataGridComboBoxColumn

The DataGridComboBoxColumn shows ordinary text initially, but provides a streamlined editing experience that allows the user to pick from a list of available options in a ComboBox control. (In fact, the user will be forced to choose from the list, as the ComboBox does not allow direct text entry.) In the example in Figure 22-8, the user is choosing the product category from a DataGridComboBoxColumn.

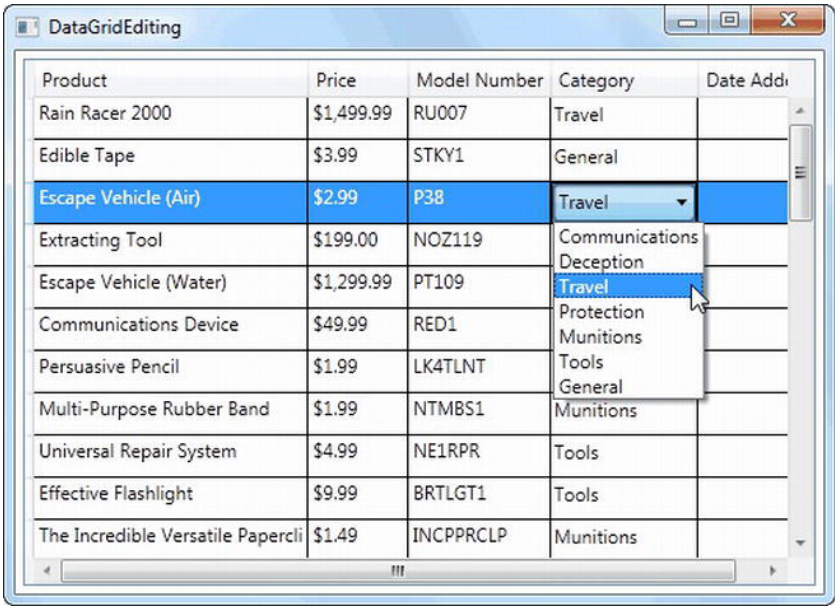

Figure 22-8. *Choosing from a list of allowed values*

To use the DataGridComboBoxColumn, you need to decide how to populate the combo box in edit mode. To do that, you simply set the DataGridComboBoxColumn.ItemsSource collection. The absolute simplest approach is to fill it by hand, in markup. The following example adds a list of strings to the combo box:

```
<DataGridComboBoxColumn Header="Category"
 SelectedItemBinding="{Binding Path=CategoryName}">
  <DataGridComboBoxColumn.ItemsSource>
    <col:ArrayList>
      <sys:String>General</sys:String>
      <sys:String>Communications</sys:String>
      <sys:String>Deception</sys:String>
      <sys:String>Munitions</sys:String>
      <sys:String>Protection</sys:String>
      <sys:String>Tools</sys:String>
      <sys:String>Travel</sys:String>
    </col:ArrayList>
  </DataGridComboBoxColumn.ItemsSource>
</DataGridComboBoxColumn>
```

In order for this markup to work as written, you must map the sys and col prefixes to the appropriate .NET namespaces:

```
<Window ...
 xmlns:col="clr-namespace:System.Collections;assembly=mscorlib"
 xmlns:sys="clr-namespace:System;assembly=mscorlib">
```

This works perfectly well, but it's not the best design, as it embeds data details deep into your user interface markup. Fortunately, you have several other options:

- Pull the data collection out of a resource. It's up to you whether you want to define the collection by using markup (as in the previous example) or generate it in code (as in the following example).
- Pull the ItemsSource collection out of a static method, using the Static markup extension. But for solid code design, limit yourself to calling a method in your window class, not one in a data class.
- Pull the data collection out of an ObjectProvider resource, which can then call a data access class.
- Set the DataGridComboBox.Column property directly in code.

In many situations, the values you display in the list aren't the values you want to store in the data object. One common case occurs when dealing with related data (for example, orders that link to products, billing records that link to customers, and so on).

The StoreDB example includes one such relationship, between products and categories. In the back-end database, each product is linked to a specific category by using the CategoryID field. This fact is hidden in the simplified data model that all the examples have used so far, which gives the Product class a CategoryName property (rather than a CategoryID property). The advantage of this approach is convenience, as it keeps the salient information—the category name for each product—close at hand. The disadvantage is that the CategoryName property isn't really editable, and there's no straightforward way to change a product from one category into another.

The following example considers a more realistic case, where each Product includes a CategoryID property. On its own, the CategoryID number doesn't mean much to the application user. To display the category name instead, you need to rely on one of several possible techniques: you can add an additional CategoryName property to the Product class (which works, but is a bit clumsy), you can use a data converter in your CategoryID bindings (which could look up the matching category name in a cached list), or you can display the CategoryID column with the DataGridComboBoxColumn (which is the approach demonstrated next).

Using this approach, instead of a list of simple strings, you bind an entire list of Category objects to the DataGridComboBoxColumn.ItemsSource property:

```
categoryColumn.ItemsSource = App.StoreDB.GetCategories();
gridProducts.ItemsSource = App.StoreDB.GetProducts();
```

You then configure the DataGridComboBoxColumn. You must set three properties:

```
<DataGridComboBoxColumn Header="Category" x:Name="categoryColumn"
 DisplayMemberPath="CategoryName" SelectedValuePath="CategoryID"
 SelectedValueBinding="{Binding Path=CategoryID}"></DataGridComboBoxColumn>
```

DisplayMemberPath tells the column which text to extract from the Category object and display in the list. SelectedValuePath tells the column what *data* to extract from the Category object. SelectedValueBinding specifies the linked field in the Product object.

DataGridTemplateColumn

The DataGridTemplateColumn uses a data template, which works in the same way as the data-template features you explored with list controls earlier. The only difference in the DataGridTemplateColumn is that it allows you to define two templates: one for data display (the CellTemplate) and one for data editing (the

CellEditingTemplate), which you'll consider shortly. Here's an example that uses the template data column to place a thumbnail image of each product in the grid (see Figure 22-9):

```
<DataGridTemplateColumn>
  <DataGridTemplateColumn.CellTemplate>
    <DataTemplate>
      <Image Stretch="None" Source=
       "{Binding Path=ProductImagePath, Converter={StaticResource ImagePathConverter}}">
      </Image>
    </DataTemplate>
  </DataGridTemplateColumn.CellTemplate>
</DataGridTemplateColumn>
```

This example assumes you've added the ImagePathConverter value converter to the UserControl.Resources collection:

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

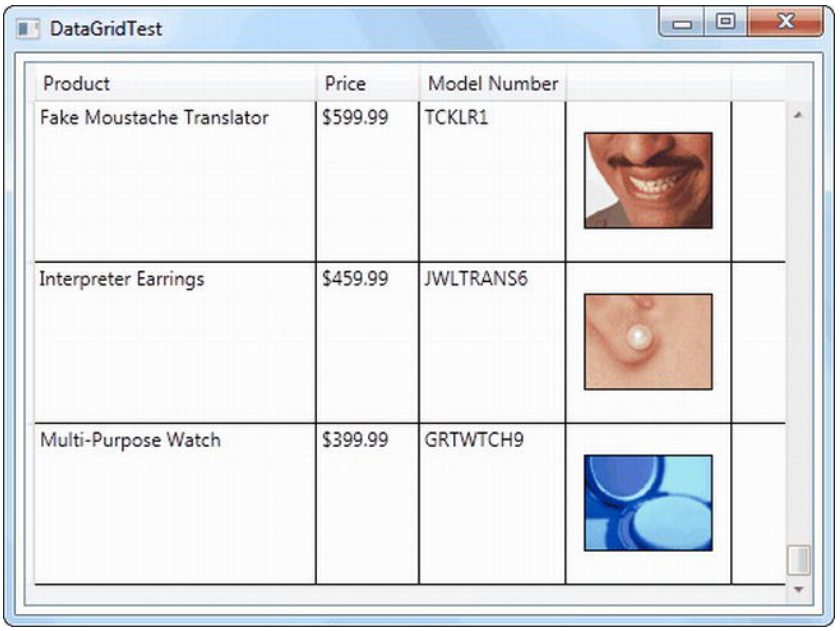

Figure 22-9. *A DataGrid with image content*

Formatting and Styling Columns

You can format a DataGridTextColumn in the same way that you format a TextBlock element, by setting the Foreground, FontFamily, FontSize, FontStyle, and FontWeight properties. However, the DataGridTextColumn doesn't expose all the properties of the TextBlock. For example, there's no way to set the often-used Wrapping property if you want to create a column that shows multiple lines of text. In this case, you need to use the ElementStyle property instead.

Essentially, the ElementStyle property lets you create a style that is applied to the element inside the DataGrid cell. In the case of a simple DataGridTextColumn, that's a TextBlock. In a DataGridCheckBoxColumn, it's a check box. In a DataGridTemplateColumn, it's whatever element you've created in the data template.

Here's a simple style that allows the text in a column to wrap:

```
<DataGridTextColumn Header="Description" Width="400"
 Binding="{Binding Path=Description}">
  <DataGridTextColumn.ElementStyle>
    <Style TargetType="TextBlock">
      <Setter Property="TextWrapping" Value="Wrap"></Setter>
    </Style>
  </DataGridTextColumn.ElementStyle>
</DataGridTextColumn>
```

To see the wrapped text, you must expand the row height. Unfortunately, the DataGrid can't size itself as flexibly as WPF layout containers can. Instead, you're forced to set a fixed row height by using the DataGrid.RowHeight property. This height applies to all rows, regardless of the amount of content they contain. Figure 22-10 shows an example with the row height set to 70 units.

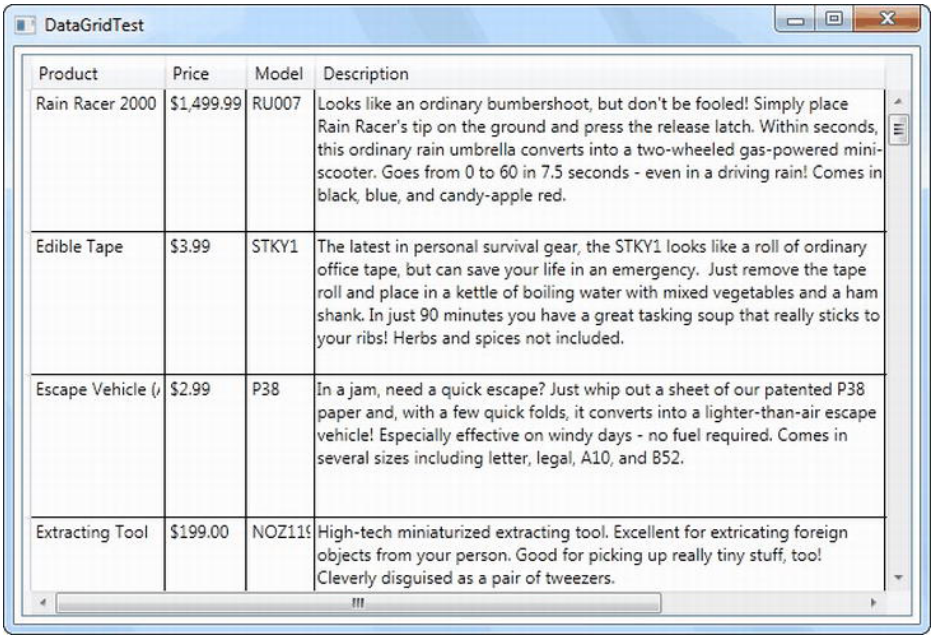

Figure 22-10. *A DataGrid with wrapped text*

■ **Tip** If you want to apply the same style to multiple columns (for example, to deal with wrappable text in several places), you can define the style in the Resources collection and then refer to it in each column by using a StaticResource.

You can use EditingElementStyle to style the element that's employed when you're editing a column. In the case of DataGridTextColumn, the editing element is the TextBox control.

The ElementStyle, ElementEditingStyle, and column properties give you a way to format all the cells in a specific column. However, in some cases, you might want to apply formatting settings to every cell in every column. The simplest way to do so is to configure a style for the DataGrid.RowStyle property. The DataGrid also exposes a small set of additional properties that allow you to format other parts of the grid, such as the column headers and row headers. Table 22-2 has the full story.

Table 22-2. *Style-Based DataGrid Properties*

Property	Style Applies To…
ColumnHeaderStyle	The TextBlock that's used for the column headers at the top of the grid
RowHeaderStyle	The TextBlock that's used for the row headers
DragIndicatorStyle	The TextBlock that's used for a column header when the user is dragging it to a new position
RowStyle	The TextBlock that's used for ordinary rows (rows in columns that haven't been expressly customized through the ElementStyle property of the column)

Formatting Rows

By setting the properties of the DataGrid column objects, you can control how entire columns are formatted. But in many cases, it's more useful to flag rows that contain specific data. For example, you may want to draw attention to high-priced products or expired shipments. You can apply this sort of formatting programmatically by handling the DataGrid.LoadingRow event.

The LoadingRow event is a powerful tool for row formatting. It gives you access to the data object for the current row, allowing you to perform simple range checks, comparison, and more-complex manipulations. It also provides the DataGridRow object for the row, letting you format the row with different colors or a different font. However, you can't format just a single cell in that row—for that, you need DataGridTemplateColumn and a custom value converter.

The LoadingRow event fires once for each row when it appears onscreen. The advantage of this approach is that your application is never forced to format the whole grid; instead, LoadingRow fires only for the rows that are currently visible. But there's also a downside. As the user scrolls through the grid, the LoadingRow event is triggered continuously. As a result, you can't place time-consuming code in the LoadingRow method unless you want scrolling to grind to a halt.

There's also another consideration: item container recycling. To lower its memory overhead, the DataGrid reuses the same DataGridRow objects to show new data as you scroll through the data. (That's why the event is called LoadingRow rather than CreatingRow.) If you're not careful, the DataGrid can load data into an already-formatted DataGridRow. To prevent this from happening, you must explicitly restore each row to its initial state.

In the following example, high-priced items are given a bright orange background (see Figure 22-11). Regular-price items are given the standard white background:

```
// Reuse brush objects for efficiency in large data displays.
private SolidColorBrush highlightBrush = new SolidColorBrush(Colors.Orange);
private SolidColorBrush normalBrush = new SolidColorBrush(Colors.White);

private void gridProducts_LoadingRow(object sender, DataGridRowEventArgs e)
{
```

```
    // Check the data object for this row.
    Product product = (Product)e.Row.DataContext;

    // Apply the conditional formatting.
    if (product.UnitCost > 100)
    {
        e.Row.Background = highlightBrush;
    }
    else
    {
        // Restore the default white background. This ensures that used,
        // formatted DataGrid objects are reset to their original appearance.
        e.Row.Background = normalBrush;
    }
}
```

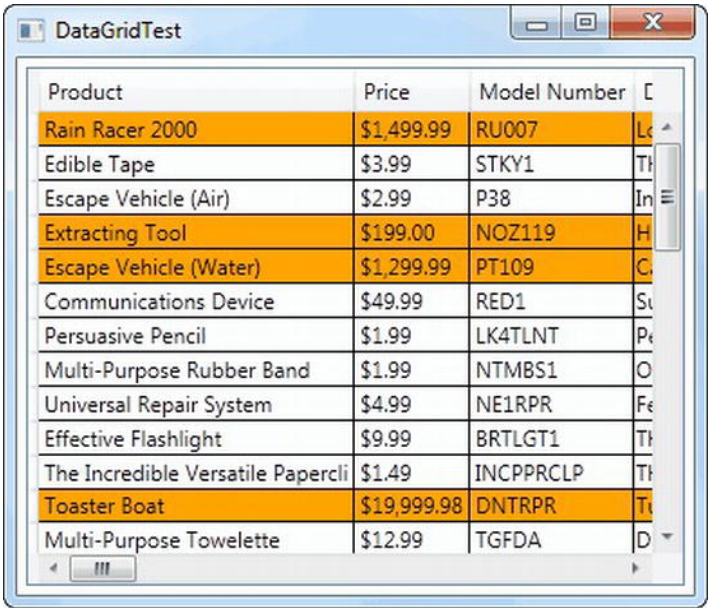

Figure 22-11. *Highlighting rows*

Remember, you have another option for performing value-based formatting: you can use a value converter that examines bound data and converts it to something else. This technique is especially powerful when combined with a DataGridTemplateColumn. For example, you can create a template-based column that contains a TextBlock, and bind the TextBlock.Background property to a value converter that sets the color based on the price. Unlike the LoadingRow approach shown previously, this technique allows you to format just the cell that contains the price, rather than the whole row. For more information about this technique, refer to Chapter 20.

■ **Note** The formatting you apply in the LoadingRow event handler applies only when the row is loaded. If you edit a row, this LoadingRow code doesn't fire (at least, not until you scroll the row out of view and then back into sight).

Displaying Row Details

The DataGrid also supports *row details*—an optional, separate display area that appears just under the column values for a row. The row-details area adds two things that you can't get from columns alone:

- It spans the full width of the DataGrid and isn't carved into separate columns, which gives you more space to work with.
- You can configure the row-details area so that it appears only for the selected row, allowing you to tuck the extra details out of the way when they're not needed.

Figure 22-12 shows a DataGrid that uses both of these behaviors. The row-details area displays the wrapped product description text, and it's shown only for the currently selected product.

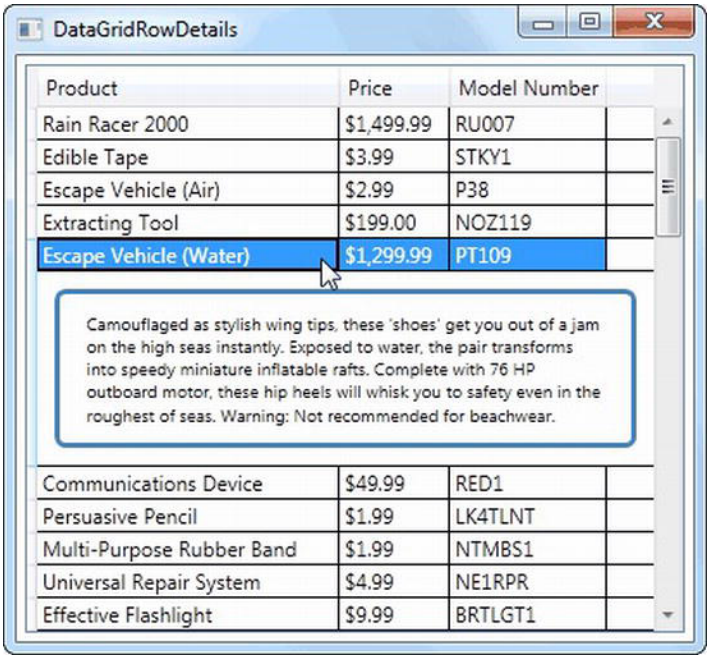

Figure 22-12. *Using the row-details area*

To create this example, you need to first define the content that's shown in the row-details area by setting the DataGrid.RowDetailsTemplate property. In this case, the row-details area uses a basic template that includes a TextBlock that shows the full product text and adds a border around it:

```
<DataGrid.RowDetailsTemplate>
  <DataTemplate>
    <Border Margin="10" Padding="10" BorderBrush="SteelBlue" BorderThickness="3"
     CornerRadius="5">
      <TextBlock Text="{Binding Path=Description}" TextWrapping="Wrap"
       FontSize="10">
      </TextBlock>
    </Border>
  </DataTemplate>
</DataGrid.RowDetailsTemplate>
```

Other options include adding controls that allow you to perform various tasks (for example, getting more information about a product, adding it to a shopping list, editing it, and so on).

You can configure the display behavior of the row-details area by setting the DataGrid.RowDetailsVisibilityMode property. By default, this property is set to VisibleWhenSelected, which means the row-details area is shown when the row is selected. Alternatively, you can set it to Visible, which means the details area of every row will be shown at once. Or you can use Collapsed, which means the details area won't be shown for any row—at least, not until you change the RowDetailsVisibilityMode in code (for example, when the user selects a certain type of row).

Freezing Columns

A *frozen* column stays in place at the left side of the DataGrid, even as you scroll to the right. Figure 22-13 shows how a frozen Product column remains visible during scrolling. Notice how the horizontal scrollbar extends under only the scrollable columns, not the frozen columns.

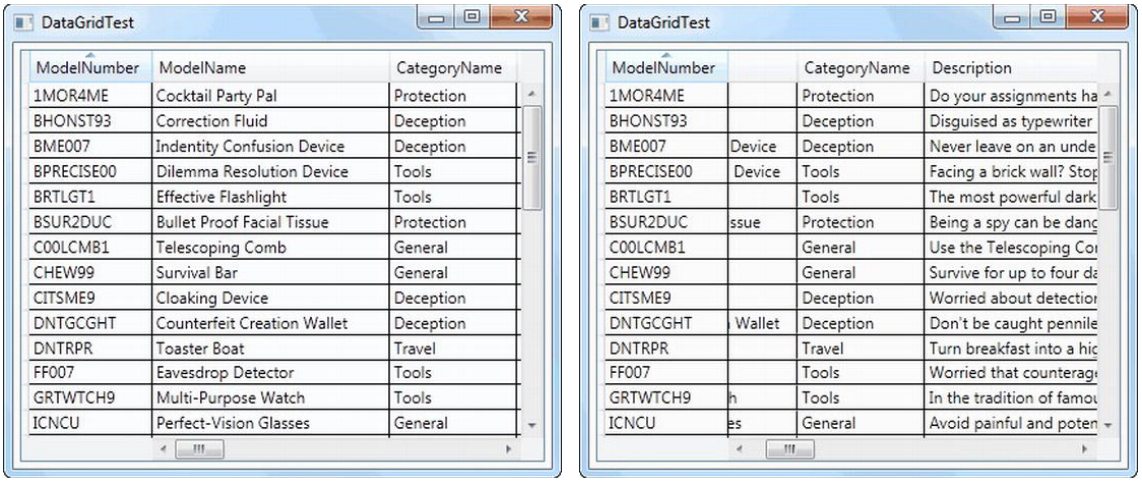

Figure 22-13. *Freezing the Product column*

Column freezing is a useful feature for very wide grids, especially when you want to make sure certain information (such as the product name or a unique identifier) is always visible. To use it, you set the DataGrid.FrozenColumnCount property to a number greater than 0. For example, a value of 1 freezes just the first column:

```
<DataGrid x:Name="gridProducts" Margin="5" AutoGenerateColumns="False"
 FrozenColumnCount="1">
```

Frozen columns must always be on the left side of the grid. If you freeze one column, it is the leftmost column; if you freeze two columns, they will be the first two on the left; and so on.

Selection

Like an ordinary list control, the DataGrid lets the user select individual items. You can react to the SelectionChanged event when this happens. To find out which data object is currently selected, you can use the SelectedItem property. If you want the user to be able to select multiple rows, set the SelectionMode property to Extended. (Single is the only other option and the default.) To select multiple rows, the user must hold down the Shift or Ctrl key. You can retrieve the collection of selected items from the SelectedItems property.

■ **Tip** You can set the selection programmatically by using the SelectedItem property. If you're setting the selection to an item that's not currently in view, it's a good idea to follow up with a call to the DataGrid.ScrollIntoView() method, which forces the DataGrid to scroll forward or backward until the item you've indicated is visible.

Sorting

The DataGrid features built-in sorting as long as you're binding a collection that implements IList (such as the List<T> and ObservableCollection<T> collections). If you meet this requirement, your DataGrid gets basic sorting for free.

To use the sorting, the user needs to click a column header. Clicking once sorts the column in ascending order based on its data type (for example, numbers are sorted from 0 up, and letters are sorted alphabetically). Click the column again, and the sort order is reversed. An arrow appears at the far-right side of the column header, indicating that the DataGrid is sorted based on the values in this column. The arrow points up for an ascending sort and down for a descending sort.

Users can sort based on multiple columns by holding down Shift while they click. For example, if you hold down Shift and click the Category column followed by the Price column, products are sorted into alphabetical category groups, and the items in each category group are ordered by price.

Ordinarily, the DataGrid sorting algorithm uses the bound data that appears in the column, which makes sense. However, you can choose a different property from the bound data object by setting a column's SortMemberPath. And if you have a DataGridTemplateColumn, you need to use SortMemberPath, because there's no Binding property to provide the bound data. If you don't, your column won't support sorting.

You can also disable sorting by setting the CanUserSortColumns property to false (or turn it off for specific columns by setting the column's CanUserSort property).

Editing

One of the DataGrid's greatest conveniences is its support for editing. A DataGrid cell switches into edit mode when the user double-clicks it. But the DataGrid lets you restrict this editing ability in several ways:

DataGrid.IsReadOnly: When this property is true, users can't edit anything.

DataGridColumn.IsReadOnly: When this property is true, users can't edit any of the values in that column.

Read-only properties: If your data object has a property with no property setter, the DataGrid is intelligent enough to notice this detail and disable column editing, just as if you had set DataGridColumn.IsReadOnly to true. Similarly, if your property isn't a simple text, numeric, or date type, the DataGrid makes it read-only (although you can remedy this situation by switching to the DataGridTemplateColumn, as described shortly).

What happens when a cell switches into edit mode depends on the column type. A DataGridTextColumn shows a text box (although it's a seamless-looking text box that fills the entire cell and has no visible border). A DataGridCheckBox column shows a check box that you can check or uncheck. But the DataGridTemplateColumn is by far the most interesting. It allows you to replace the standard editing text box with a more specialized input control.

For example, the following column shows a date. When the user double-clicks to edit that value, it turns into a drop-down DatePicker (see Figure 22-14) with the current value preselected:

```
<DataGridTemplateColumn Header="Date Added">
  <DataGridTemplateColumn.CellTemplate>
    <DataTemplate>
      <TextBlock Margin="4" Text=
 "{Binding Path=DateAdded, Converter={StaticResource DateOnlyConverter}}">
      </TextBlock>
    </DataTemplate>
  </DataGridTemplateColumn.CellTemplate>
  <DataGridTemplateColumn.CellEditingTemplate>
    <DataTemplate>
      <DatePicker SelectedDate="{Binding Path=DateAdded, Mode=TwoWay}">
      </DatePicker>
    </DataTemplate>
  </DataGridTemplateColumn.CellEditingTemplate>
</DataGridTemplateColumn>
```

Figure 22-14. *Editing dates with the DatePicker*

The DataGrid automatically supports the same basic validation system you learned about in the previous chapter, which reacts to problems in the data-binding system (such as the inability to convert supplied text to the appropriate data type) or exceptions thrown by the property setter. Here's an example that uses a custom validation rule to validate the UnitCost field:

```
<DataGridTextColumn Header="Price">
  <DataGridTextColumn.Binding>
    <Binding Path="UnitCost" StringFormat="{}{0:C}">
      <Binding.ValidationRules>
        <local:PositivePriceRule Max="999.99" />
      </Binding.ValidationRules>
    </Binding>
  </DataGridTextColumn.Binding>
</DataGridTextColumn>
```

The default ErrorTemplate for the DataGridCell displays a red outline around the invalid value, much the same as other input controls such as the TextBox.

You can implement validation a couple of other ways with a DataGrid. One option is to use the DataGrid's editing events, which are listed in Table 22-3. The order of rows matches the order that the events fire in the DataGrid.

Table 22-3. *DataGrid Editing Events*

Name	Description
BeginningEdit	Occurs when the cell is about to be put in edit mode. You can examine the column and row that are currently being edited, check the cell value, and cancel this operation by using the DataGridBeginningEditEventArgs.Cancel property.
PreparingCellForEdit	Used for template columns. At this point, you can perform any last-minute initialization that's required for the editing controls. Use DataGridPreparingCellForEditEventArgs.EditingElement to access the element in the CellEditingTemplate.
CellEditEnding	Occurs when the cell is about to exit edit mode. DataGridCellEditEndingEventArgs.EditAction tells you whether the user is attempting to accept the edit (for example, by pressing Enter or clicking another cell) or cancel it (by pressing the Esc key). You can examine the new data and set the Cancel property to roll back an attempted change.
RowEditEnding	Occurs when the user navigates to a new row after editing the current row. As with CellEditEnding, you can use this point to perform validation and cancel the change. Typically, you'll perform validation that involves several columns—for example, ensuring that the value in one column isn't greater than the value in another.

If you need a place to perform validation logic that is specific to your page (and so can't be baked into the data objects), you can write custom validation logic that responds to the CellEditEnding and RowEditEnding events. Check column rules in the CellEditEnding event handler, and validate the consistency of the entire row in the RowEditEnding event. And remember that if you cancel an edit, you should provide an explanation of the problem (usually in a TextBlock elsewhere on the page).

The Last Word

In this chapter, you took a closer look at the ItemsControl classes provided by WPF. You learned how to use the ListView to create lists with multiple viewing modes, the TreeView to show hierarchical data, and the DataGrid to view and edit a dense assortment of data in a single place.

The most impressive aspect of all these classes is that they derive from a single base class—the ItemsControl—that defines their essential functionality. The fact that all these controls share the same content model, the same data-binding ability, and the same styling and templating features is one of WPF's small miracles. Remarkably, the ItemsControl defines all the basics for any WPF list control, even those that wrap hierarchical data, such as the TreeView. The only change in the model is that the children of these controls (TreeViewItem objects) are *themselves* ItemsControl objects, with the ability to host their own children.