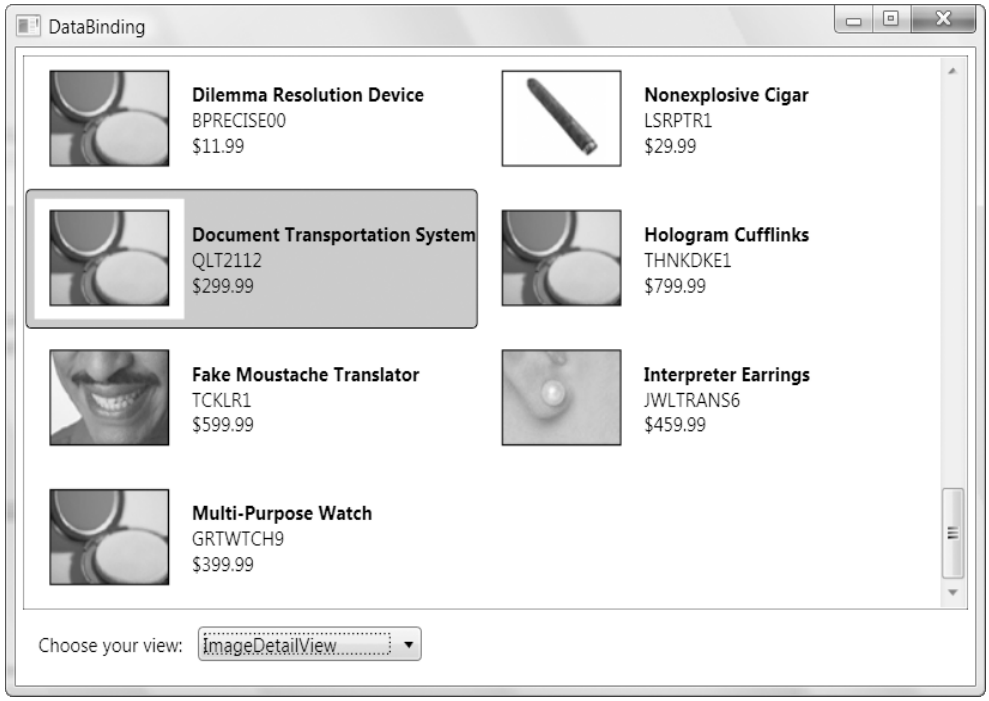

Figure 22-4. *A detailed image view*

The View Class

The first step that's required to build this example is the class representing 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 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

Once 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 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 using data templates with the HeaderTemplate or HeaderTemplateSelector property.

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.

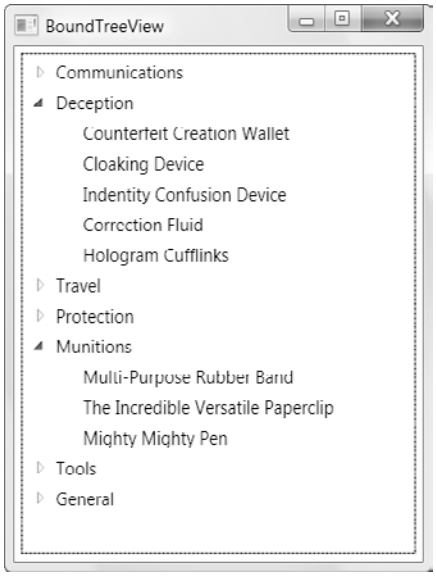

Figure 22-5. *A TreeView of categories and products*

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 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 date 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 like 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.

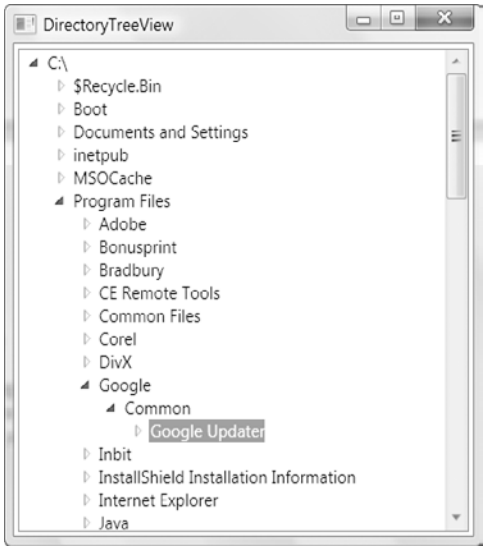

Figure 22-6. *Digging into a directory tree*

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 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 organization chart. You can view the full code at `http://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.

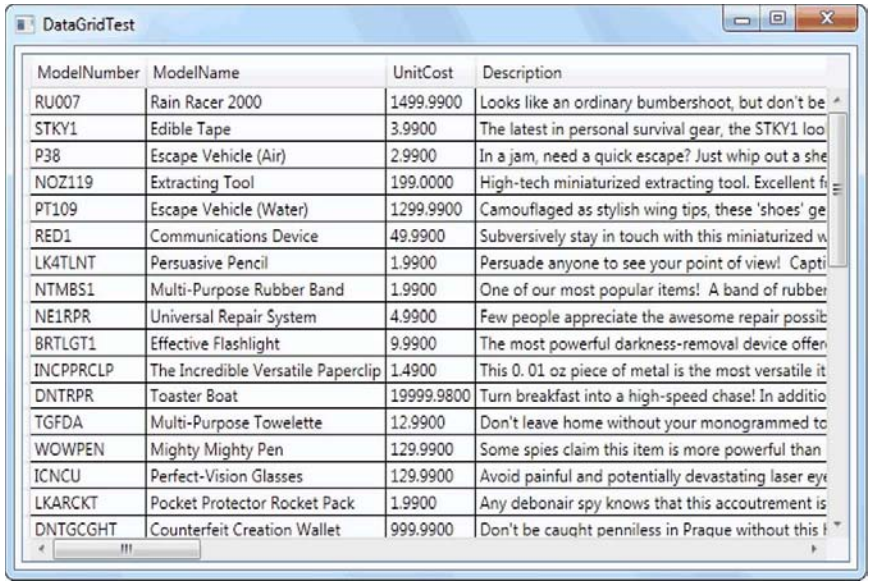

Figure 22-7. *A DataGrid with automatically generated columns*

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 shows any data. It indicates the currently selected row (with an arrow) and when the row is being edited (with an arrow in a circle).
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 (like 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 in between columns.
HorizontalGridLinesBrush	The brush that's used to paint the grid lines in 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 scroll bar 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 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 like 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 (like 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 like 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.

The 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.