The CustomDrawnElement doesn't need to contain any child content, so it derives directly from FrameworkElement. It allows only a single property to be set—the background color of the gradient. (The foreground color is hard-coded to be white, although you could easily change this detail.)

```
public class CustomDrawnElement : FrameworkElement
{
    public static DependencyProperty BackgroundColorProperty;

    static CustomDrawnElement()
    {
        FrameworkPropertyMetadata metadata =
          new FrameworkPropertyMetadata(Colors.Yellow);
        metadata.AffectsRender = true;
        BackgroundColorProperty =  DependencyProperty.Register("BackgroundColor",
        typeof(Color), typeof(CustomDrawnElement), metadata);
    }

    public Color BackgroundColor
    {
        get { return (Color)GetValue(BackgroundColorProperty); }
        set { SetValue(BackgroundColorProperty, value); }
    }
    ...
```

The BackgroundColor dependency property is specifically marked with the FrameworkPropertyMetadata.AffectRender flag. As a result, WPF will automatically call OnRender() whenever the color is changed. However, you also need to make sure OnRender() is called when the mouse moves to a new position. This is handled by calling the InvalidateVisual() method at the right times:

```
    ...
    protected override void OnMouseMove(MouseEventArgs e)
    {
        base.OnMouseMove(e);
        this.InvalidateVisual();
    }

    protected override void OnMouseLeave(MouseEventArgs e)
    {
        base.OnMouseLeave(e);
        this.InvalidateVisual();
    }
    ...
```

The only remaining detail is the rendering code. It uses the DrawingContext.DrawRectangle() method to paint the element's background. The ActualWidth and ActualHeight properties indicate the final rendered dimensions of the control.

```
    ...
    protected override void OnRender(DrawingContext dc)
    {
        base.OnRender(dc);
```

```
        Rect bounds = new Rect(0, 0, base.ActualWidth, base.ActualHeight);
        dc.DrawRectangle(GetForegroundBrush(), null, bounds);
    }
    ...
```

Finally, a private helper method named GetForegroundBrush() constructs the correct RadialGradientBrush based on the current position of the mouse. To calculate the center point, you need to convert the current position of the mouse over the element to a relative position from 0 to 1, which is what the RadialGradientBrush expects.

```
    ...
    private Brush GetForegroundBrush()
    {
        if (!IsMouseOver)
        {
            return new SolidColorBrush(BackgroundColor);
        }
        else
        {
            RadialGradientBrush brush = new RadialGradientBrush(
              Colors.White, BackgroundColor);

            // Get the position of the mouse in device-independent units,
            // relative to the control itself.
            Point absoluteGradientOrigin = Mouse.GetPosition(this);

            // Convert the point coordinates to proportional (0 to 1) values.
            Point relativeGradientOrigin = new Point(
              absoluteGradientOrigin.X / base.ActualWidth,
              absoluteGradientOrigin.Y / base.ActualHeight);

            // Adjust the brush.
            brush.GradientOrigin = relativeGradientOrigin;
            brush.Center = relativeGradientOrigin;

            return brush;
        }
    }
}
```

This completes the example.

A Custom Decorator

As a general rule, you should never use custom drawing in a control. If you do, you violate the premise of WPF's lookless controls. The problem is that once you hardwire in some drawing logic, you've ensured that a portion of your control's visual appearance cannot be customized through the control template.

A much better approach is to design a separate element that draws your custom content (such as the CustomDrawnElement class in the previous example) and then use that element inside the default control template for your control. That's the approach used in many WPF controls, and you saw it at work in the Button control in Chapter 17.

It's worth quickly considering how you can adapt the previous example so that it can function as part of a control template. Custom-drawn elements usually play two roles in a control template:

- They draw some small graphical detail (like the arrow on a scroll button).
- They provide a more detailed background or frame around another element.

The second approach requires a custom decorator. You can change the CustomDrawnElement into a custom-drawn element by making two small changes. First, derive it from Decorator:

```
public class CustomDrawnDecorator : Decorator
```

Next, override the OnMeasure() method to specify the required size. It's the responsibility of all decorators to consider their children, add the extra space required for their embellishments, and then return the combined size. The CustomDrawnDecorator doesn't need any extra space to draw a border. Instead, it simply makes itself as large as the content warrants using this code:

```
protected override Size MeasureOverride(Size constraint)
{
    UIElement child = this.Child;
    if (child != null)
    {
        child.Measure(constraint);
        return child.DesiredSize;
    }
    else
    {
        return new Size();
    }
}
```

Once you've created your custom decorator, you can use it in a custom control template. For example, here's a button template that places the mouse-tracking gradient background behind the button content. It uses template bindings to make sure the properties for alignment and padding are respected.

```
<ControlTemplate x:Key="ButtonWithCustomChrome">
  <lib:CustomDrawnDecorator BackgroundColor="LightGreen">
    <ContentPresenter Margin="{TemplateBinding Padding}"
     HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
     VerticalAlignment="{TemplateBinding VerticalContentAlignment}"
     ContentTemplate="{TemplateBinding ContentControl.ContentTemplate}"
     Content="{TemplateBinding ContentControl.Content}"
     RecognizesAccessKey="True" />
  </lib:CustomDrawnDecorator>
</ControlTemplate>
```

You can now use this template to restyle your buttons with a new look. Of course, to make your decorator more practical, you'd probably want to make it vary its appearance when the mouse button is clicked. You can do this using triggers that modify properties in your chrome class. Chapter 17 has a complete discussion of this design.

The Last Word

In this chapter, you took a detailed look at custom control development in WPF. You saw how to build basic user controls and extend existing WPF controls and how to create the WPF gold standard—a template-based lookless control. Finally, you considered custom drawing and how you can use custom-drawn content with a template-based control.

If you're planning to dive deeper into the world of custom control development, you'll find some excellent samples online. One good starting point is the set of control customization samples that are included with the .NET Framework SDK and that are available for convenient individual download at `http://code.msdn.microsoft.com/wpfsamples#controlcustomization`. Another worthwhile download is the Bag-o-Tricks sample project provided by Kevin Moore (a former program manager on the WPF team) at `http://j832.com/bagotricks`, which includes everything from basic date controls to a panel with built-in animation.

CHAPTER 19

Data Binding

Data binding is the time-honored tradition of pulling information out of an object and displaying it in your application's user interface, without writing the tedious code that does all the work. Often, rich clients use *two-way* data binding, which adds the ability to push information from the user interface back into some object—again, with little or no code. Because many Windows applications are all about data (and all of them need to deal with data some of the time), data binding is a top concern in a user interface technology like WPF.

Developers who are approaching WPF from a Windows Forms background will find that WPF data binding has many similarities. As in Windows Forms, WPF data binding allows you to create bindings that take information from just about any property of any object and stuff it into just about any property of any element. WPF also includes a set of list controls that can handle entire collections of information and allow you to navigate through them. However, there are significant changes in the way that data binding is implemented behind the scenes, some impressive new functionality, and a fair bit of tweaking and fine-tuning. Many of the same concepts apply, but the same code won't.

In this chapter, you'll learn how to use WPF data binding. You'll create declarative bindings that extract the information you need and display it in various types of elements. You'll also learn how to plug this system into a back-end database.

What's New Although data binding basics remain the same, WPF 4 has improved its support for virtualization and container recycling—two features that are critical for ensuring good performance in massively large lists. You'll learn about both in the section "Improving Performance in Large Lists."

Binding to a Database with Custom Objects

When developers hear the term *data binding*, they often think of one specific application—pulling information out of a database and showing it onscreen with little or no code.

As you saw in Chapter 8, data binding in WPF is a much more general tool. Even if your application never comes into contact with a database, it's still likely to use data binding to automate the way elements interact or translate an object model into a suitable display.

However, you can learn a lot about the details of object binding by considering a traditional example that queries and updates a table in a database. In this chapter, you'll use an example that retrieves a catalog of products. The first step in building this example is to create a custom data access component.

■ **Note** The downloadable code for this chapter includes the custom data access component and a database script that installs the sample data, so you can test all the examples. But if you don't have a test database server or you don't want to go to the trouble of creating a new database, you can use an alternate version of the data access component that's also included with the code. This version simply loads the data from a file, while still exposing the same set of classes and methods. It's perfect for testing but obviously impractical for a real application.

Building a Data Access Component

In professional applications, database code is not embedded in the code-behind class for a window but encapsulated in a dedicated class. For even better componentization, these data access classes can be pulled out of your application altogether and compiled in a separate DLL component. This is particularly true when writing code that accesses a database (because this code tends to be extremely performance-sensitive), but it's a good design no matter where your data lives.

Designing Data Access Components

No matter how you plan to use data binding (or even if you don't), your data access code should always be coded in a separate class. This approach is the only way you have the slightest chance to make sure you can efficiently maintain, optimize, troubleshoot, and (optionally) reuse your data access code.

When creating a data class, you should follow a few basic guidelines in this section:

- Open and close connections quickly. Open the database connection in every method call, and close it before the method ends. This way, a connection can't be inadvertently left open. One way to ensure the connection is closed at the appropriate time is with a using block.
- Implement error handling. Use error handling to make sure that connections are closed even if an exception occurs.
- Follow stateless design practices. Accept all the information needed for a method in its parameters, and return all the retrieved data through the return value. This avoids complications in a number of scenarios (for example, if you need to create a multithreaded application or host your database component on a server).
- Store the connection string in one place. Ideally, this is the configuration file for your application.

The database component that's shown in the following example retrieves a table of product information from the Store database, which is a sample database for the fictional IBuySpy store included with some Microsoft case studies. Figure 19-1 shows two tables in the Store database and their schemas.

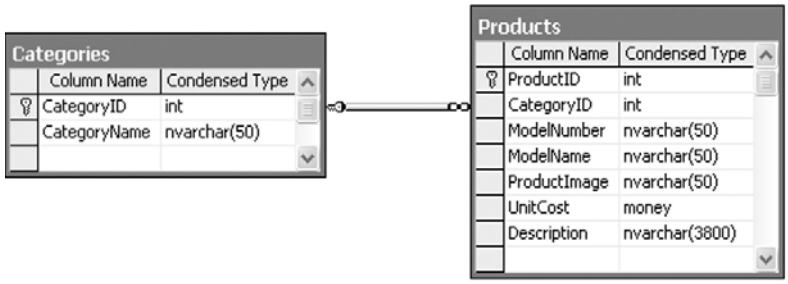

Figure 19-1. *A portion of the Store database*

The data access class is exceedingly simple—it provides just a single method that allows the caller to retrieve one product record. Here's the basic outline:

```
public class StoreDB
{
    // Get the connection string from the current configuration file.
    private string connectionString = Properties.Settings.Default.StoreDatabase;

    public Product GetProduct(int ID)
    {
        ...
    }
}
```

The query is performed through a stored procedure in the database named GetProduct. The connection string isn't hard-coded—instead, it's retrieved through an application setting in the .config file for this application. (To view or set application settings, double-click the Properties node in the Solution Explorer, and then click the Settings tab.)

When other windows need data, they call the StoreDB.GetProduct() method to retrieve a Product object. The Product object is a custom object that has a sole purpose in life—to represent the information for a single row in the Products table. You'll consider it in the next section.

You have several options for making the StoreDB class available to the windows in your application:

- The window could create an instance of StoreDB whenever it needs to access the database.
- You could change the methods in the StoreDB class to be static.
- You could create a single instance of StoreDB and make it available through a static property in another class (following the "factory" pattern).

The first two options are reasonable, but both of them limit your flexibility. The first choice prevents you from caching data objects for use in multiple windows. Even if you don't want to use that caching right away, it's worth designing your application in such a way that it's easy to implement later. Similarly, the second approach assumes you won't have any instance-specific state that you need to retain in the StoreDB class. Although this is a good design principle, you might want to retain some details (such as the connection string) in memory. If you convert the StoreDB class to use static

methods, it becomes much more difficult to access different instances of the Store database in different back-end data stores.

Ultimately, the third option is the most flexible. It preserves the switchboard design by forcing all the windows to work through a single property. Here's an example that makes an instance of StoreDB available through the Application class:

```
public partial class App : System.Windows.Application
{
    private static StoreDB storeDB = new StoreDB();
    public static StoreDB StoreDB
    {
        get { return storeDB; }
    }
}
```

In this book, we're primarily interested with how data objects can be bound to WPF elements. The actual process that deals with creating and filling these data objects (as well as other implementation details, such as whether StoreDB caches the data over several method calls, whether it uses stored procedures instead of inline queries, whether it fetches the data from a local XML file when offline, and so on) isn't our focus. However, just to get an understanding of what's taking place, here's the complete code:

```
public class StoreDB
{
    private string connectionString = Properties.Settings.Default.StoreDatabase;

    public Product GetProduct(int ID)
    {
        SqlConnection con = new SqlConnection(connectionString);
        SqlCommand cmd = new SqlCommand("GetProductByID", con);
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Parameters.AddWithValue("@ProductID", ID);

        try
        {
            con.Open();
            SqlDataReader reader = cmd.ExecuteReader(CommandBehavior.SingleRow);
            if (reader.Read())
            {
                // Create a Product object that wraps the
                // current record.
                Product product = new Product((string)reader["ModelNumber"],
                    (string)reader["ModelName"], (decimal)reader["UnitCost"],
                    (string)reader["Description"] ,
                    (string)reader["ProductImage"]);
                return(product);
            }
            else
            {
                return null;
            }
        }
```

```
        finally
        {
            con.Close();
        }
    }
}
```

■ **Note** Currently, the GetProduct() method doesn't include any exception handling code, so all exceptions will bubble up the calling code. This is a reasonable design choice, but you might want to catch the exception in GetProduct(), perform cleanup or logging as required, and then rethrow the exception to notify the calling code of the problem. This design pattern is called *caller inform*.

Building a Data Object

The data object is the information package that you plan to display in your user interface. Any class works, provided it consists of public properties (fields and private properties aren't supported). In addition, if you want to use this object to make changes (via two-way binding), the properties cannot be read-only.

Here's the Product object that's used by StoreDB:

```
public class Product
{
    private string modelNumber;
    public string ModelNumber
    {
        get { return modelNumber; }
        set { modelNumber = value; }
    }

    private string modelName;
    public string ModelName
    {
        get { return modelName; }
        set { modelName = value; }
    }

    private decimal unitCost;
    public decimal UnitCost
    {
        get { return unitCost; }
        set { unitCost = value; }
    }

    private string description;
    public string Description
    {
        get { return description; }
```

```
        set { description = value; }
    }

    public Product(string modelNumber, string modelName,
      decimal unitCost, string description)
    {
        ModelNumber = modelNumber;
        ModelName = modelName;
        UnitCost = unitCost;
        Description = description;
    }
}
```

Displaying the Bound Object

The final step is to create an instance of the Product object and then bind it to your controls. Although you could create a Product object and store it as a resource or a static property, neither approach makes much sense. Instead, you need to use StoreDB to create the appropriate object at runtime and then bind that to your window.

■ **Note** Although the declarative no-code approach sounds more elegant, there are plenty of good reasons to mix a little code into your data-bound windows. For example, if you're querying a database, you probably want to handle the connection in your code so that you can decide how to handle exceptions and inform the user of problems.

Consider the simple window shown in Figure 19-2. It allows the user to supply a product code, and it then shows the corresponding product in the Grid in the lower portion of the window.

Figure 19-2. *Querying a product*

When you design this window, you don't have access to the Product object that will supply the data at runtime. However, you can still create your bindings without indicating the data source. You simply need to indicate the property that each element uses from the Product class.

Here's the full markup for displaying a Product object:

```
<Grid Name="gridProductDetails">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition Height="*"></RowDefinition>
  </Grid.RowDefinitions>

  <TextBlock Margin="7">Model Number:</TextBlock>
  <TextBox Margin="5" Grid.Column="1"
   Text="{Binding Path=ModelNumber}"></TextBox>
  <TextBlock Margin="7" Grid.Row="1">Model Name:</TextBlock>
  <TextBox Margin="5" Grid.Row="1" Grid.Column="1"
   Text="{Binding Path=ModelName}"></TextBox>
  <TextBlock Margin="7" Grid.Row="2">Unit Cost:</TextBlock>
  <TextBox Margin="5" Grid.Row="2" Grid.Column="1"
   Text="{Binding Path=UnitCost}"></TextBox>
  <TextBlock Margin="7,7,7,0" Grid.Row="3">Description:</TextBlock>
  <TextBox Margin="7" Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="2"
   TextWrapping="Wrap" Text="{Binding Path=Description}"></TextBox>
</Grid>
```

Notice that the Grid wrapping all these details is given a name so that you can manipulate it in code and complete your data bindings.

When you first run this application, no information will appear. Even though you've defined your bindings, no source object is available.

When the user clicks the button at runtime, you use the StoreDB class to get the appropriate product data. Although you could create each binding programmatically, this wouldn't make much sense (and it wouldn't save much code over just populating the controls by hand). However, the DataContext property provides a perfect shortcut. If you set it for the Grid that contains all your data binding expressions, all your binding expressions will use it to fill themselves with data.

Here's the event handling code that reacts when the user clicks the button:

```
private void cmdGetProduct_Click(object sender, RoutedEventArgs e)
{
    int ID;
    if (Int32.TryParse(txtID.Text, out ID))
    {
        try
        {
            gridProductDetails.DataContext = App.StoreDB.GetProduct(ID);
        }
```

```
        catch
        {
            MessageBox.Show("Error contacting database.");
        }
    }
    else
    {
        MessageBox.Show("Invalid ID.");
    }
}
```

Binding With Null Values

The current Product class assumes that it will get a full complement of product data. However, database tables frequently include nullable fields, where a null value represents missing or inapplicable information. You can reflect this reality in your data classes by using nullable data types for simple value types like numbers and dates. For example, in the Product class, you can use `decimal?` instead of `decimal`. Of course, reference types, such as strings and full-fledged objects, always support null values.

The results of binding a null value are predictable: the target element shows nothing at all. For numeric fields, this behavior is useful because it distinguishes between a missing value (in which case the element shows nothing) and a zero value (in which case it shows the text "0"). However, it's worth noting that you can change how WPF handles null values by setting the TargetNullValue property in your binding expression. If you do, the value you supply will be displayed whenever the data source has a null value. Here's an example that shows the text "[No Description Provided]" when the Product.Description property is null:

```
Text="{Binding Path=Description, TargetNullValue=[No Description Provided]}"
```

The square brackets around the TargetNullValue text are optional. In this example, they're intended to help the user recognize that the displayed text isn't drawn from the database.

Updating the Database

You don't need to do anything extra to enable data object updates with this example. The TextBox.Text property uses two-way binding by default, which means that the bound Product object is modified as you edit the text in the text boxes. (Technically, each property is updated when you tab to a new field, because the default source update mode for the TextBox.Text property is LostFocus. To review the different update modes that binding expressions support, refer to Chapter 8.)

You can commit changes to the database at any time. All you need is to add an UpdateProduct() method to the StoreDB class and an Update button to the window. When clicked, your code can grab the current Product object from the data context and use it to commit the update:

```
private void cmdUpdateProduct_Click(object sender, RoutedEventArgs e)
{
    Product product = (Product)gridProductDetails.DataContext;
    try
    {
        App.StoreDB.UpdateProduct(product);
    }
    catch
    {
        MessageBox.Show("Error contacting database.");
    }
}
```

This example has one potential stumbling block. When you click the Update button, the focus changes to that button, and any uncommitted edit is applied to the Product object. However, if you set the Update button to be a default button (by setting IsDefault to true), there's another possibility. A user could make a change in one of the fields and hit Enter to trigger the update process without committing the last change. To avoid this possibility, you can explicitly force the focus to change before you execute any database code, like this:

```
FocusManager.SetFocusedElement(this, (Button)sender);
```

Change Notification

The Product binding example works so well because each Product object is essentially fixed—it never changes (except if the user edits the text in one of the linked text boxes).

For simple scenarios, where you're primarily interested in displaying content and letting the user edit it, this behavior is perfectly acceptable. However, it's not difficult to imagine a different situation, where the bound Product object might be modified elsewhere in your code. For example, imagine an Increase Price button that executes this line of code:

```
product.UnitCost *= 1.1M;
```

■ **Note** Although you could retrieve the Product object from the data context, this example assumes you're also storing it as a member variable in your window class, which simplifies your code and requires less type casting.

When you run this code, you'll find that even though the Product object has been changed, the old value remains in the text box. That's because the text box has no way of knowing that you've changed a value.

You can use three approaches to solve this problem:

- You can make each property in the Product class a dependency property using the syntax you learned about in Chapter 4. (In this case, your class must derive from DependencyObject.) Although this approach gets WPF to do the work for you (which is nice), it makes the most sense in elements—classes that have a visual appearance in a window. It's not the most natural approach for data classes like Product.
- You can raise an event for each property. In this case, the event must have the name *PropertyName*Changed (for example, UnitCostChanged). It's up to you to fire the event when the property is changed.
- You can implement the System.ComponentModel.INotifyPropertyChanged interface, which requires a single event named PropertyChanged. You must then raise the PropertyChanged event whenever a property changes and indicate which property has changed by supplying the property name as a string. It's still up to you to raise this event when a property changes, but you don't need to define a separate event for each property.

The first approach relies on the WPF dependency property infrastructure, while both the second and the third rely on events. Usually, when creating a data object, you'll use the third approach. It's the simplest choice for non-element classes.

■ **Note** You can actually use one other approach. If you suspect a change has been made to a bound object and that bound object doesn't support change notifications in any of the proper ways, you can retrieve the BindingExpression object (using the FrameworkElement.GetBindingExpression() method) and call BindingExpression.UpdateTarget() to trigger a refresh. Obviously, this is the most awkward solution—you can almost see the duct tape that's holding it together.

Here's the definition for a revamped Product class that uses the INotifyPropertyChanged interface, with the code for the implementation of the PropertyChanged event:

```
public class Product : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null)
          PropertyChanged(this, e);
    }
}
```

Now you simply need to fire the PropertyChanged event in all your property setters:

```
private decimal unitCost;
public decimal UnitCost
{
    get { return unitCost; }
    set {
        unitCost = value;
        OnPropertyChanged(new PropertyChangedEventArgs("UnitCost"));
    }
}
```

If you use this version of the Product class in the previous example, you'll get the behavior you expect. When you change the current Product object, the new information will appear in the text box immediately.

■ **Tip** If several values have changed, you can call OnPropertyChanged() and pass in an empty string. This tells WPF to reevaluate the binding expressions that are bound to any property in your class.

Binding to a Collection of Objects

Binding to a single object is quite straightforward. But life gets more interesting when you need to bind to some collection of objects—for example, all the products in a table.

Although every dependency property supports the single-value binding you've seen so far, collection binding requires an element with a bit more intelligence. In WPF, all the classes that derive from ItemsControl have the ability to show an entire list of items. Data binding possibilities include the ListBox, ComboBox, ListView, and DataGrid (and the Menu and TreeView for hierarchical data).

■ **Tip** Although it seems like WPF offers a relatively small set of list controls, these controls allow you to show your data in a virtually unlimited number of ways. That's because the list controls support data templates, which allow you to control exactly how items are displayed. You'll learn more about data templates in Chapter 20.

To support collection binding, the ItemsControl class defines the three key properties listed in Table 19-1.

Table 19-1. *Properties in the ItemsControl Class for Data Binding*

Name	Description
ItemsSource	Points to the collection that has all the objects that will be shown in the list.
DisplayMemberPath	Identifies the property that will be used to create the display text for each item.
ItemTemplate	Accepts a data template that will be used to create the visual appearance of each item. This property is far more powerful than DisplayMemberPath, and you'll learn how to use it in Chapter 20.

At this point, you're probably wondering exactly what type of collections you can stuff in the ItemSource property. Happily, you can use just about anything. All you need is support for the IEnumerable interface, which is provided by arrays, all types of collections, and many more specialized objects that wrap groups of items. However, the support you get from a basic IEnumerable interface is limited to read-only binding. If you want to edit the collection (for example, you want to allow inserts and deletions), you need a bit more infrastructure, as you'll see shortly.

Displaying and Editing Collection Items

Consider the window shown in Figure 19-3, which shows a list of products. When you choose a product, the information for that product appears in the bottom section of the window, where you can edit it. (In this example, a GridSplitter lets you adjust the space given to the top and bottom portions of the window.)

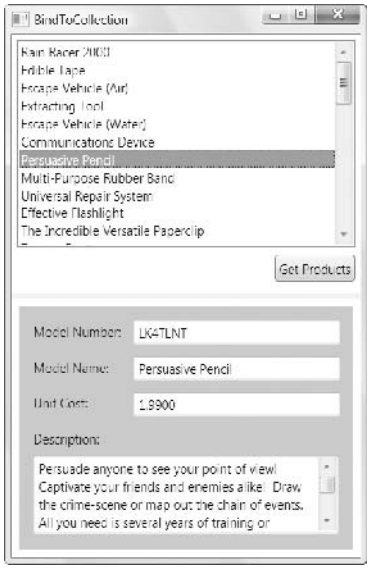

Figure 19-3. *A list of products*

To create this example, you need to begin by building your data access logic. In this case, the StoreDB.GetProducts() method retrieves the list of all the products in the database using the GetProducts stored procedure. A Product object is created for each record and added to a generic List collection. (You could use any collection here—for example, an array or a weakly typed ArrayList would work equivalently.)

Here's the GetProducts() code:

```
public List<Product> GetProducts()
{
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand("GetProducts", con);
    cmd.CommandType = CommandType.StoredProcedure;

    List<Product> products = new List<Product>();
    try
    {
        con.Open();
        SqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
            // Create a Product object that wraps the
            // current record.
            Product product = new Product((string)reader["ModelNumber"],
              (string)reader["ModelName"], (decimal)reader["UnitCost"],
              (string)reader["Description"], (string)reader["CategoryName"],
              (string)reader["ProductImage"]);

            // Add to collection
            products.Add(product);
        }
    }
    finally
    {
        con.Close();
    }
    return products;
}
```

When the Get Products button is clicked, the event handling code calls the GetProducts() method and supplies it as the ItemsSource for list. The collection is also stored as a member variable in the window class for easier access elsewhere in your code.

```
private List<Product> products;

private void cmdGetProducts_Click(object sender, RoutedEventArgs e)
{
    products = App.StoreDB.GetProducts();
    lstProducts.ItemsSource = products;
}
```

This successfully fills the list with Product objects. However, the list doesn't know how to display a product object, so it will simply call the ToString() method. Because this method hasn't been overridden

in the Product class, this has the unimpressive result of showing the fully qualified class name for every item (see Figure 19-4).

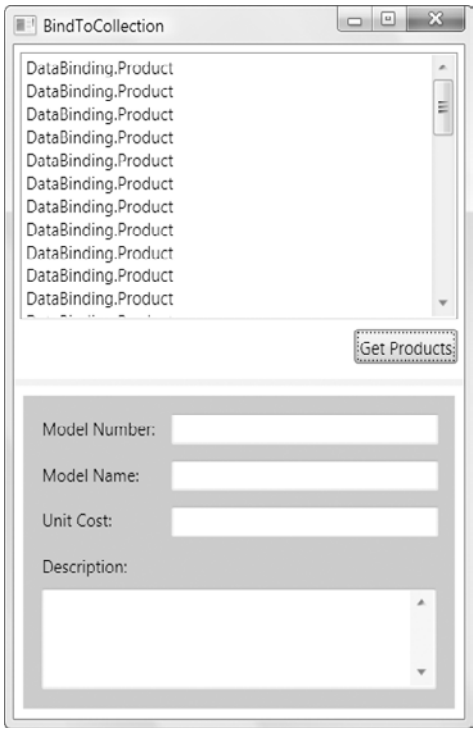

Figure 19-4. *An unhelpful bound list*

You have three options to solve this problem:

- Set the DisplayMemberPath property of the list. For example, set this to ModelName to get the result shown in Figure 19-4.
- Override the ToString() method to return more useful information. For example, you could return a string with the model number and model name of each item. This approach gives you a way to show more than one property in the list (for example, it's great for combining the FirstName and LastName properties in a Customer class). However, you still don't have much control over how the data is presented.
- Supply a data template. This way, you can show any arrangement of property values (along with fixed text). You'll learn how to use this trick in Chapter 20.

Once you've decided how to display information in the list, you're ready to move on to the second challenge: displaying the details for the currently selected item in the grid that appears below the list.

You could handle this challenge by responding to the SelectionChanged event and manually changing the data context of the grid, but there's a quicker approach that doesn't require any code. You simply need to set a binding expression for the Grid.DataContent property that pulls the selected Product object out of the list, as shown here:

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

When the window first appears, nothing is selected in the list. The ListBox.SelectedItem property is null, and therefore the Grid.DataContext is too, and no information appears. As soon as you select an item, the data context is set to the corresponding object, and all the information appears.

If you try this example, you'll be surprised to see that it's already fully functional. You can edit product items, navigate away (using the list), and then return to see that your edits were successfully committed. In fact, you can even change a value that affects the display text in the list. If you modify the model name and tab to another control, the corresponding entry in the list is refreshed automatically. (Experienced developers will recognize this as a frill that Windows Forms applications lacked.)

■ **Tip** To prevent a field from being edited, set the IsLocked property of the text box to true, or, better yet, use a read-only control like a TextBlock.

Master-Details Display

As you've seen, you can bind other elements to the SelectedItem property of your list to show more details about the currently selected item. Interestingly, you can use a similar technique to build a master-details display of your data. For example, you can create a window that shows a list of categories and a list of products. When the user chooses a category in the first list, you can show just the products that belong to that category in the second list.

To pull this off, you need to have a *parent* data object that provides a collection of related *child* data objects through a property. For example, you could build a Category product that provides a property named Category.Products with the products that belong to that category. (In fact, you can find an example of a Category class that's designed like this in Chapter 21.) You can then build a master-details display with two lists. Fill your first list with Category objects. To show the related products, bind your second list—the list that displays products—to the SelectedItem.Products property of the first list. This tells the second list to grab the current Category object, extract its collection of linked Product objects, and display them.

You can find an example that uses related data in Chapter 21, with a TreeView that shows a categorized list of products.

Of course, to complete this example, from an application perspective you'll need to supply some code. For example, you might need an UpdateProducts() method that accepts your collection or products and executes the appropriate statements. Because an ordinary .NET object doesn't provide any change tracking, this is a situation where you might want to consider using the ADO.NET DataSet (as described a little later in this chapter). Alternatively, you might want to force users to update records one at a time. (One option is to disable the list when text is modified in a text box and force the user to then cancel the change by clicking Cancel or apply it immediately by clicking Update.)

Inserting and Removing Collection Items

One limitation of the previous example is that it won't pick up changes you make to the collection. It notices changed Product objects, but it won't update the list if you add a new item or remove one through code.

For example, imagine you add a Delete button that executes this code:

```
private void cmdDeleteProduct_Click(object sender, RoutedEventArgs e)
{
    products.Remove((Product)lstProducts.SelectedItem);
}
```

The deleted item is removed from the collection, but it remains stubbornly visible in the bound list.

To enable collection change tracking, you need to use a collection that implements the INotifyCollectionChanged interface. Most generic collections don't, including the List collection used in the current example. In fact, WPF includes a single collection that uses INotifyCollectionChanged: the ObservableCollection class.

■ **Note** If you have an object model that you're porting over from the Windows Forms world, you can use the Windows Forms equivalent of ObservableCollection, which is BindingList. The BindingList collection implements IBindingList instead of INotifyCollectionChanged, which includes a ListChanged event that plays the same role as the INotifyCollectionChanged.CollectionChanged event.

You can derive a custom collection from ObservableCollection to customize the way it works, but that's not necessary. In the current example, it's enough to replace the List<Product> object with an ObservableCollection<Product>, as shown here:

```
public List<Product> GetProducts()
{
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand("GetProducts", con);
    cmd.CommandType = CommandType.StoredProcedure;

    ObservableCollection<Product> products = new ObservableCollection<Product>();
    ...
```