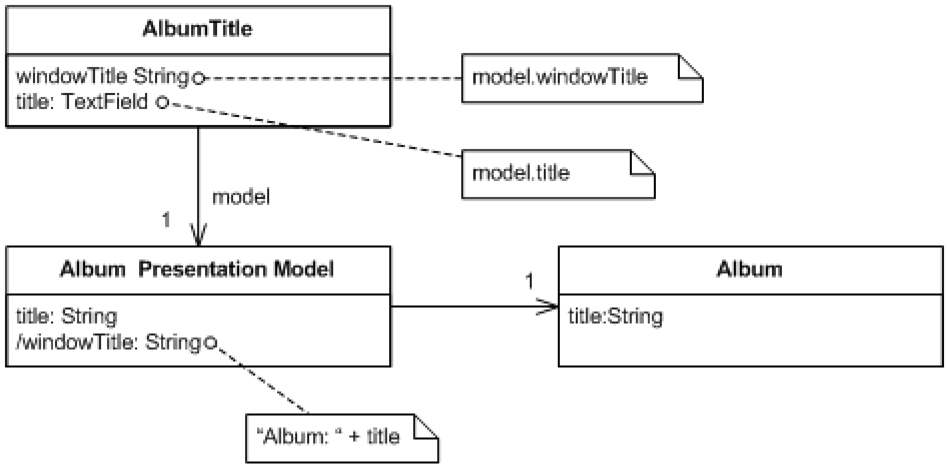
Presentation Model

Represent the state and behavior of the presentation independently of the GUI controls used in the interface



Also Known as: **Application Model**

Martin Fowler

19 July 2004

This is part of the Further Enterprise Application Architecture development writing that I was doing in the mid 2000's. Sadly too many other things have claimed my attention since, so I haven't had time to work on them further, nor do I see much time in the foreseeable future. As such this material is very much in draft form and I won’t be doing any corrections or updates until I’m able to find time to work on it again.

GUIs consist of widgets that contain the state of the GUI screen. Leaving the state of the GUI in widgets makes it harder to get at this state, since that involves manipulating widget APIs, and also encourages putting presentation behavior in the view class.

Presentation Model pulls the state and behavior of the view out into a model class that is part of the presentation. The Presentation Model coordinates with the domain layer and provides an interface to the view that minimizes decision making in the view. The view either stores all its state in the Presentation Model or synchronizes its state with Presentation Model frequently

Presentation Model may interact with several domain objects, but Presentation Model is not a GUI friendly facade to a specific domain object. Instead it is easier to consider Presentation Model as an abstract of the view that is not dependent on a specific GUI framework. While several views can utilize the same Presentation Model, each view should require only one Presentation Model. In the case of composition a Presentation Model may contain one or many child Presentation Model instances, but each child control will also have only one Presentation Model.

Presentation Model is known to users of Visual Works Smalltalk as **Application Model**

How it Works

The essence of a Presentation Model is of a fully self-contained class that represents all the data and behavior of the UI window, but without any of the controls used to render that UI on the screen. A view then simply projects the state of the presentation model onto the glass.

To do this the Presentation Model will have data fields for all the dynamic information of the view. This won't just include the contents of controls, but also things like whether or not they are enabled. In general the Presentation Model does not need to hold all of this control state (which would be lot) but any state that may change during the interaction of the user. So if a field is always enabled, there won't be extra data for its state in the Presentation Model.

Since the Presentation Model contains data that the view needs to display the controls you need to synchronize the Presentation Model with the view. This synchronization usually needs to be tighter than synchronization with the domain - screen synchronization is not sufficient, you'll need field or key synchronization.

To illustrate things a bit better, I'll use the aspect of the running example where the composer field is only enabled if the classical check box is checked.

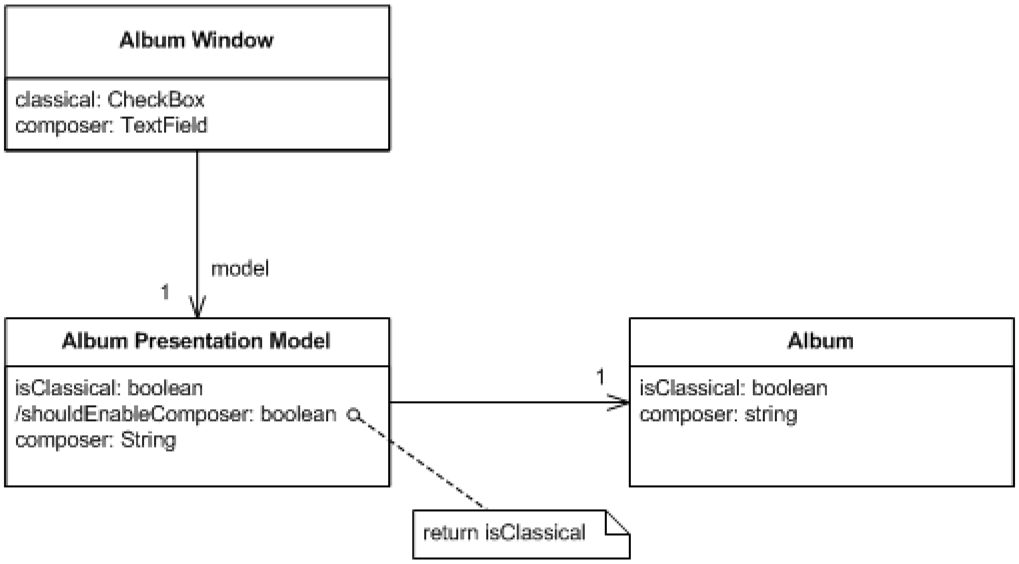


Figure 1: Classes showing structure relevant to clicking the classical check box

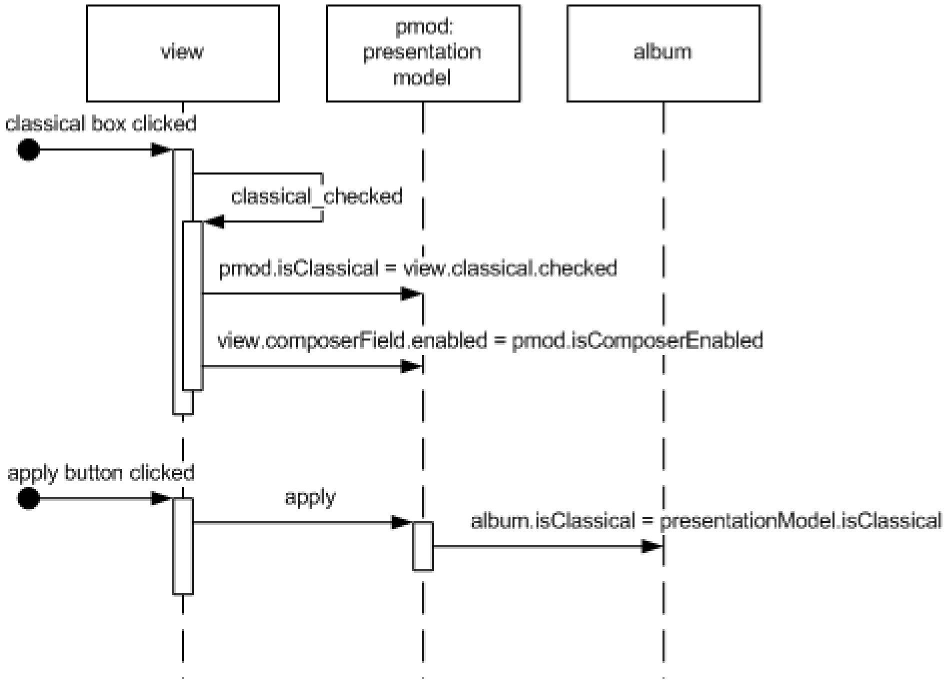

view
pmod: presentation model
album
classical box clicked
classical_checked
pmod.isClassical = view.classical.checked
view.composerField.enabled = pmod.isComposerEnabled
apply button clicked
apply
album.isClassical = presentationModel.isClassical

Figure 2: How objects react to clicking the classical check box.

When someone clicks the classical check box the check box changes its state and then calls the appropriate event handler in the view. This event handler saves the state of the view to Presentation Model and then updates itself from the Presentation Model

(I'm assuming a coarse-grained synchronization here.) The Presentation Model contains the logic that says that the composer field is only enabled if the check box is checked, so the when the view updates itself from the Presentation Model, the composer field control changes its enablement state. I've indicated on the diagram that the Presentation Model would typically have a property specifically to mark whether the composer field should be enabled. This will, of course, just return the value of the isClassical property in this case - but the separate property is important because that property encapsulates how the Presentation Model determines whether the composer field is enabled - clearly indicating that that decision is the responsibility of the Presentation Model.

This small example is illustrates the essence of the idea of the Presentation Model - all the decisions needed for presentation display are made by the Presentation Model leaving the view to be utterly simple.

Probably the most annoying part of Presentation Model is the synchronization between Presentation Model and view. It's simple code to write, but I always like to minimize this kind of boring repetitive code. Ideally some kind of framework could handle this, which I'm hoping will happen some day with technologies like .NET's data binding.

A particular decision you have to make with synchronization in Presentation Model is which class should contain the synchronization code. Often, this decision is largely based on the desired level of test coverage and the chosen implementation of Presentation Model. If you put the synchronization in the view, it won't get picked up by tests on the Presentation Model. If you put it in the Presentation Model you add a dependency to the view in the Presentation Model which means more coupling and stubbing. You could add a mapper between them, but adds yet more classes to coordinate. When making the decision of which implementation to use it is important to remember that although faults do occur in synchronization code, they are usually easy to spot and fix (unless you use fine-grained synchronization).

An important implementation detail of Presentation Model is whether the View should reference the Presentation Model or the Presentation Model should reference the View. Both implementations provide pros and cons.

A Presentation Model that references a view generally maintains the synchronization code in the Presentation Model. The resulting view is very dumb. The view contains setters for any state that is dynamic and raises events in response to user actions. The views implement interfaces allowing for easy stubbing when testing the Presentation Model. The Presentation Model will observe the view and respond to events by changing any appropriate state and reloading the entire view. As a result the synchronization code can be easily tested without needing the actual UI class.

A Presentation Model that is referenced by a view generally maintains the synchronization code in the view. Because the synchronization code is generally easy to write and easy to spot errors it is recommended that the testing occur on the Presentation Model and not the View. If you are compelled to write tests for the view this should be a clue that the view contains code that should belong in the Presentation Model. If you prefer to test the synchronization, a Presentation Model that references a view implementation is recommended.

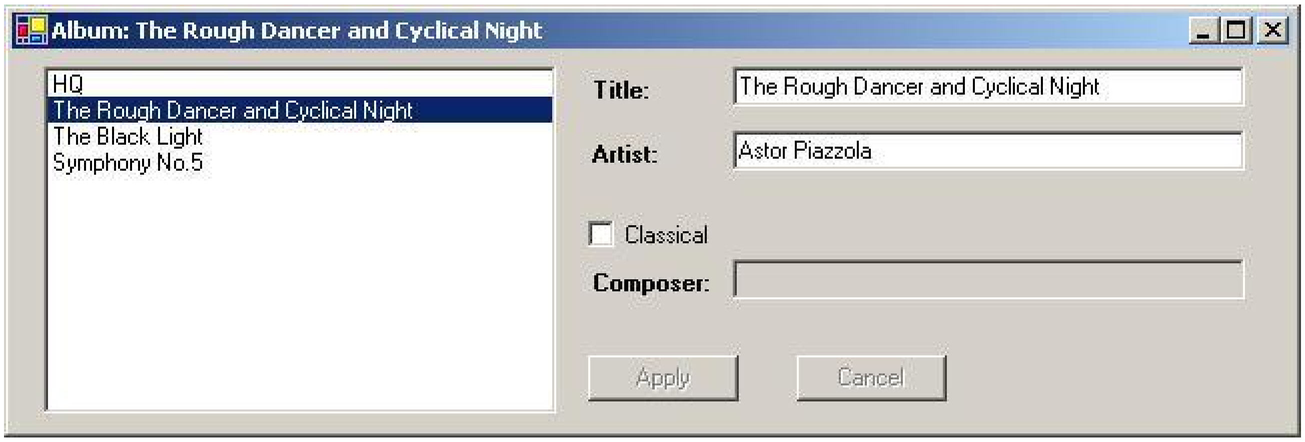
When to Use It

Presentation Model is a pattern that pulls presentation behavior from a view. As such it's an alternative to to Supervising Controller and Passive View. It's useful for allowing you to test without the UI, support for some form of multiple view and a separation of concerns which may make it easier to develop the user interface.

Compared to Passive View and Supervising Controller, Presentation Model allows you to write logic that is completely independent of the views used for display. You also do not need to rely on the view to store state. The downside is that you need a synchronization mechanism between the presentation model and the view. This synchronization can be very simple, but it is required. Separated Presentation requires much less synchronization and Passive View doesn't need any at all.

Example: Running Example (View References PM) (C#)

Here's a version of the running example, developed in C# with Presentation Model.


Album: The Rough Dancer and Cyclical Night

HQ
The Rough Dancer and Cyclical Night
The Black Light
Symphony No.5

Title: The Rough Dancer and Cyclical Night
Artist: Astor Piazzola
Classical
Composer:

Figure 3: The album window.

I'll start discussing the basic layout from the domain model outwards. Since the domain isn't the focus of this example, it's very uninteresting. It's essentially just a data set with a single table holding the data for an album. Here's the code for setting up a few test albums. I'm using a strongly typed data set.

```
public static DsAlbum AlbumDataSet() {
  DsAlbum result = new DsAlbum();
  result.Albums.AddAlbumsRow(1, "HQ", "Roy Harper", false, null);
  result.Albums.AddAlbumsRow(2, "The Rough Dancer and Cyclical Night", "Astor Piazzola", false, null)
  result.Albums.AddAlbumsRow(3, "The Black Light", "Calexico", false, null);
  result.Albums.AddAlbumsRow(4, "Symphony No.5", "CBSO", true, "Sibelius" );
  result.AcceptChanges();
  return result;
}
```

The Presentation Model wraps this data set and provides properties to get at the data. There's a single instance of the Presentation Model for the whole table, corresponding to the single instance of the window. The Presentation Model has fields for the data set and also keeps track of which album is currently selected.

class PmodAlbum...

```
  public PmodAlbum(DsAlbum albums) {
    this._data = albums;
    _selectedAlbumNumber = 0;
  }
  private DsAlbum _data;
  private int _selectedAlbumNumber;
```

PmodAlbum provides properties to get at the data in the data set. Essentially I provide a property for each bit of information that the form needs to display. For those values which are just pulled directly out of the data set, this property is pretty simple.

class PmodAlbum...

```
  public String Title {
    get {return SelectedAlbum.Title;}
    set {SelectedAlbum.Title = value;}
  }
  public String Artist {
    get {return SelectedAlbum.Artist;}
    set {SelectedAlbum.Artist = value;}
  }
  public bool IsClassical {
    get {return SelectedAlbum.IsClassical;}
    set {SelectedAlbum.IsClassical = value;}
  }
  public String Composer {
    get {
      return (SelectedAlbum.IsComposerNull()) ? "" : SelectedAlbum.Composer;
    }
    set {
      if (IsClassical) SelectedAlbum.Composer = value;
    }
  }
  public DsAlbum.AlbumsRow SelectedAlbum {
    get {return Data.Albums[SelectedAlbumNumber];}
  }
```

The title of the window is based on the album title. I provide this through another property.

class PmodAlbum...

```
  public String FormTitle
  {
    get {return "Album: " + Title;}
  }
```

I have a property to see if the composer field should be enabled.

class PmodAlbum...

```
  public bool IsComposerFieldEnabled {
    get {return IsClassical;}
  }
```

This is just a call to the public IsClassical property. You may wonder why the form doesn't just call this directly - but this is the essence of the encapsulation that the Presentation Model provides. PmodAlbum decides what the logic is for enabling that field, the fact that it's simply based on a property is known to the Presentation Model but not to the view.

The apply and cancel buttons should only be enabled if the data has changed. I can provide this by checking the state of that row of the dataset, since data sets record this information.

class PmodAlbum...

```
  public bool IsApplyEnabled {
    get {return HasRowChanged;}
  }
  public bool IsCancelEnabled {
    get {return HasRowChanged;}
  }
  public bool HasRowChanged {
    get {return SelectedAlbum.RowState == DataRowState.Modified;}
  }
```

The list box in the view shows a list of the album titles. PmodAlbum provides this list.

class PmodAlbum...

```
  public String[] AlbumList {
    get {
      String[] result = new String[Data.Albums.Rows.Count];
      for (int i = 0; i < result.Length; i++)
        result[i] = Data.Albums[i].Title;
      return result;
    }
  }
```

So that covers the interface that PmodAlbum presents to the view. Next I'll look at how I do the synchronization between the view and the Presentation Model. I've put the synchronization methods in the view and am using coarse-grained synchronization. First I have a method to push the state of the view into the Presentation Model.

class FrmAlbum...

```
  private void SaveToPmod() {
    model.Artist = txtArtist.Text;
    model.Title = txtTitle.Text;
    model.IsClassical = chkClassical.Checked;
    model.Composer = txtComposer.Text;
  }
```

This method is very simple, just assigning the mutable parts of the view to the Presentation Model. The load method is a touch more complicated.

class FrmAlbum...

```
  private void LoadFromPmod() {
    if (NotLoadingView) {
      _isLoadingView = true;
      lstAlbums.DataSource = model.AlbumList;
      lstAlbums.SelectedIndex = model.SelectedAlbumNumber;
      txtArtist.Text = model.Artist;
      txtTitle.Text = model.Title;
      this.Text = model.FormTitle;
      chkClassical.Checked = model.IsClassical;
      txtComposer.Enabled = model.IsComposerFieldEnabled;
      txtComposer.Text = model.Composer;
      btnApply.Enabled = model.IsApplyEnabled;
      btnCancel.Enabled = model.IsCancelEnabled;
      _isLoadingView = false;
    }
  }
  private bool _isLoadingView = false;
  private bool NotLoadingView {
    get {return !_isLoadingView;}
  }
private void SyncWithPmod() {
  if (NotLoadingView) {
    SaveToPmod();
    LoadFromPmod();
  }
}
```

The complication here is avoiding a infinite recursion since synchronizing causes fields on the form to update which triggers synchronization.... I guard against that with a flag.

With these synchronization methods in place, the next step is just to call the right bit of synchronization in event handlers for the controls. Most of the time this easy, just call `SyncWithPmod` when data changes.

class FrmAlbum...

```
  private void txtTitle_TextChanged(object sender, System.EventArgs e){
    SyncWithPmod();
  }
```

Some cases are more involved. When the user clicks on a new item in the list we need to navigate to a new album and show its data.

class FrmAlbum...

```
  private void lstAlbums_SelectedIndexChanged(object sender, System.EventArgs e){
    if (NotLoadingView) {
      model.SelectedAlbumNumber = lstAlbums.SelectedIndex;
      LoadFromPmod();
    }
  }
```

class PmodAlbum...

```
  public int SelectedAlbumNumber {
    get {return _selectedAlbumNumber;}
    set {
      if (_selectedAlbumNumber != value) {
        Cancel();
        _selectedAlbumNumber = value;
      }
    }
  }
```

Notice that this method abandons any changes if you click on the list. I've done this awful bit of usability to keep the example simple, the form should really at least pop up a confirmation box to avoid losing the changes.

The apply and cancel buttons delegate what to do to the Presentation Model.

class FrmAlbum...

```
  private void btnApply_Click(object sender, System.EventArgs e)    {
    model.Apply();
    LoadFromPmod();
  }
  private void btnCancel_Click(object sender, System.EventArgs e){
    model.Cancel();
    LoadFromPmod();
  }
```

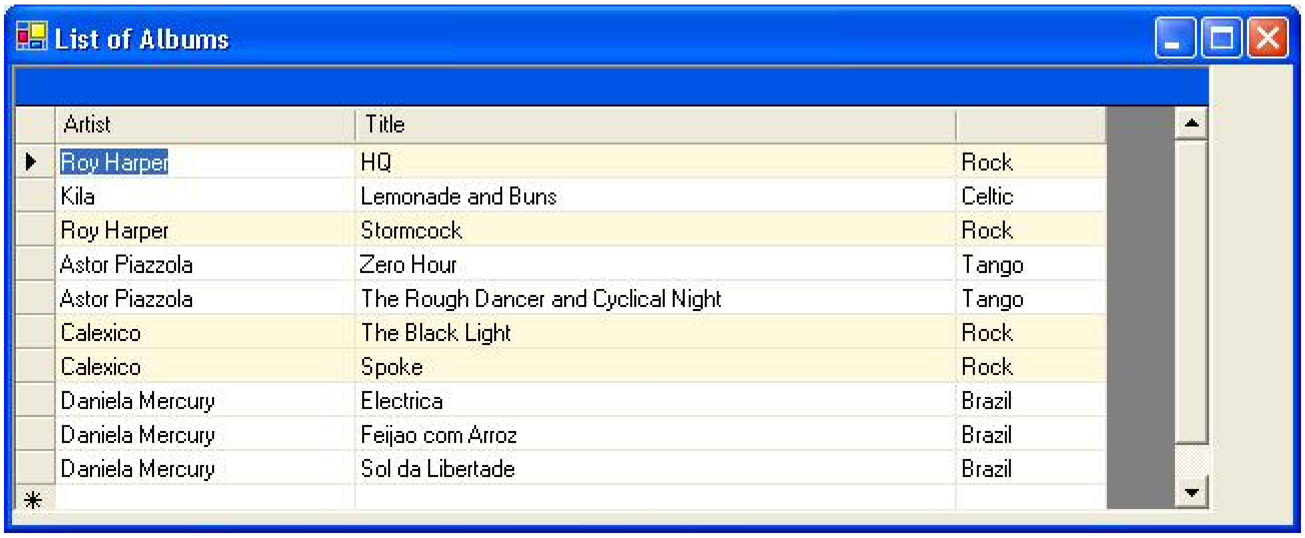
class PmodAlbum...

```
  public void Apply ()    {
    SelectedAlbum.AcceptChanges();
  }
  public void Cancel() {
    SelectedAlbum.RejectChanges();
  }
```

So although I can move most of the behavior to the Presentation Model, the view still retains some intelligence. For the testing aspect of Presentation Model to work better, it would be nice to move more. Certainly you can move more into the Presentation Model by moving the synchronization logic there, at the expense of having the Presentation Model know more about the view.

Example: Data Binding Table Example (C#)

As I first looked at Presentation Model in the .NET framework, it seemed that data binding provided excellent technology to make Presentation Model work simply. So far limitations in the current version of data binding holds it back from places that I'm sure it will eventually go. One area where data binding can work very well is with read-only data, so here is an example that shows this as well as how tables can fit in with a Presentation Model design.



Artist	Title	
Roy Harper	HQ	Rock
Kila	Lemonade and Buns	Celtic
Roy Harper	Stormcock	Rock
Astor Piazzola	Zero Hour	Tango
Astor Piazzola	The Rough Dancer and Cyclical Night	Tango
Calexico	The Black Light	Rock
Calexico	Spoke	Rock
Daniela Mercury	Electrica	Brazil
Daniela Mercury	Feijao com Arroz	Brazil
Daniela Mercury	Sol da Libertade	Brazil

Figure 4: A list of albums with the rock ones highlighted.

This is just a list of albums. The extra behavior is that each rock album should have it's row colored in cornsilk.

I'm using a slightly different data set to the other example. Here is the code for some test data.

```
public static AlbumList AlbumGridDataSet()
{
  AlbumList result = new AlbumList();
  result.Albums.AddAlbumsRow(1, "HQ", "Roy Harper", "Rock");
  result.Albums.AddAlbumsRow(2, "Lemonade and Buns", "Kila", "Celtic");
  result.Albums.AddAlbumsRow(3, "Stormcock", "Roy Harper", "Rock");
  result.Albums.AddAlbumsRow(4, "Zero Hour", "Astor Piazzola", "Tango");
  result.Albums.AddAlbumsRow(5, "The Rough Dancer and Cyclical Night", "Astor Piazzola", "Tango");
  result.Albums.AddAlbumsRow(6, "The Black Light", "Calexico", "Rock");
  result.Albums.AddAlbumsRow(7, "Spoke", "Calexico", "Rock");
  result.Albums.AddAlbumsRow(8, "Electrica", "Daniela Mercury", "Brazil");
  result.Albums.AddAlbumsRow(9, "Feijao com Arroz", "Daniela Mercury", "Brazil");
  result.Albums.AddAlbumsRow(10, "Sol da Libertade", "Daniela Mercury", "Brazil");
  Console.WriteLine(result);
  return result;
}
```

The presentation model in this case reveals its internal data set as a property. This allows the form to data bind directly to the cells in the data set.

```
private AlbumList _dsAlbums;
internal AlbumList DsAlbums {
  get {return _dsAlbums;}
}
```

To support the highlighting, the presentation model provides an additional method that indexes into the table.

```
internal Color RowColor(int row) {
  return (Albums[row].genre.Equals("Rock")) ? Color.Cornsilk : Color.White;
}
private AlbumList.AlbumsDataTable Albums {
  get {return DsAlbums.Albums;}
}
```

This method is similar to the ones in a simple example, the difference being that methods on table data need cell coordinates to pick out parts of the table. In this case all we need is a row number, but in general we may need row and column numbers.

From here on I can use the standard data binding facilities that come with visual studio. I can bind table cells easily to data in the data set, and also to data on the Presentation Model.

Getting the color to work is a little bit more involved. This is straying a little bit away from the main thrust of the example, but the whole thing gets its complication because there's no way to have row by row highlighting on the standard WinForms table control. In general the answer to this need is to buy a third party control, but I'm too cheap to do this. So for the curious here's what I did (the idea was mostly ripped off from http://www.syncfusion.com/FAQ/WinForms/). I'm going to assume you're familiar with the guts of WinForms from now on.

Essentially I made a subclass of `DataGridTextBoxColumn` which adds the color highlighting behavior. You link up the new behavior by passing in a delegate that handles the behavior.

class ColorableDataGridTextBoxColumn...

```
  public ColorableDataGridTextBoxColumn (ColorGetter getcolorRowCol, DataGridTextBoxColumn original)
  {
    _delGetColor = getcolorRowCol;
    copyFrom(original);
  }
  public delegate Color ColorGetter(int row);
  private ColorGetter _delGetColor;
```

The constructor takes the original DataGridTextBoxColumn as well as the delegate. What I'd really like to do here is to use the decorator pattern to wrap the original but the original, like so many classes in WinForms, is all sealed up. So instead I copy over all the properties of the original into my subclass. This won't work is there are vital properties that can't be copied because you can't read or write to them. It seems to work here for now.

class ColorableDataGridTextBoxColumn...

```
  void copyFrom (DataGridTextBoxColumn original) {
    PropertyInfo[] props = original.GetType().GetProperties();
    foreach (PropertyInfo p in props) {
      if (p.CanWrite && p.CanRead)
        p.SetValue(this, p.GetValue(original, null), null) ;
    }
  }
```

Fortunately the paint method is virtual (otherwise I would need a whole new data grid.) I can use it to insert the appropriate background color using the delegate.

class ColorableDataGridTextBoxColumn...

```
protected override void Paint(System.Drawing.Graphics g, System.Drawing.Rectangle bounds,
  System.Windows.Forms.CurrencyManager source, int rowNum,
  System.Drawing.Brush backBrush, System.Drawing.Brush foreBrush,
  bool alignToRight)
{
  base.Paint(g, bounds, source, rowNum, new SolidBrush(_delGetColor(rowNum)), foreBrush, alignToRig
}
```

To put this new table in place, I replace the columns of the data table in the page load after the controls have been built on the form.

class FrmAlbums...

```
private void FrmAlbums_Load(object sender, System.EventArgs e){
  bindData();
  replaceColumnStyles();
}
private void replaceColumnStyles() {
  ColorableDataGridTextBoxColumn.ReplaceColumnStyles(dgsAlbums,
    new ColorableDataGridTextBoxColumn.ColorGetter(model.RowColor));
}
```

class ColorableDataGridTextBoxColumn...

```
public static void ReplaceColumnStyles(DataGridTableStyle grid, ColorGetter del) {
  for (int i = 0; i < grid.GridColumnStyles.Count; i++) {
    DataGridTextBoxColumn old = (DataGridTextBoxColumn) grid.GridColumnStyles[0];
    grid.GridColumnStyles.RemoveAt(0);
    grid.GridColumnStyles.Add(new ColorableDataGridTextBoxColumn(del, old));
  }
}
```

It works, but I'll admit it's a lot more messy than I would like. If I were doing this for real, I'd want to look into a third party control. However I've seen this done in a production system and it worked just fine.