SILVERLIGHT PATTERNS

Model-View-ViewModel In Silverlight 2 Apps

Shawn Wildermuth

CODE DOWNLOAD AVAILABLE FROM THE MSDN CODE GALLERY

Browse the Code Online

THIS ARTICLE DISCUSSES:

- Silverlight 2 development
- Model-View-ViewModel pattern
- Views and the View Model
- Concessions to Silverlight 2

THIS ARTICLE USES THE FOLLOWING TECHNOLOGIES:

Silverlight 2, Visual Studio

Contents

Now that Silverlight 2 has been released, the number of applications built on top of it is growing, and with that comes some growing pains. The basic structure supported by the Silverlight 2 template implies a tight integration between the user interface (UI) and any data that you are working with. While this tight integration is useful for learning the technology, it becomes a hindrance to testing, refactoring, and maintenance. I will show you how to separate the UI from the data by using mature patterns for application design.

The Problem

The core of the problem is tight coupling, which is the result of mixing the layers of your application together. When one layer has intimate knowledge about how another layer does its job, then your application is tightly coupled. Take a simple data entry application that lets you query for homes for sale in a particular city. In a tightly coupled application, you could define the query to perform the search in a button handler in your user interface. As the schema changes or the semantics of the search change, both the data layer and the user interface layer have to be updated.

This presents a problem in code quality and complexity. Every time the data layer changes, you have to synchronize and test the application to be sure that the changes are not breaking changes. When everything is bound tightly together, any movement in one part of the application can cause rippling changes across the rest of the code. When you are creating something simple in Silverlight2, like a movie player or a menu widget, tightly coupling the application's components is not likely to be an issue. As the size of a project increases, however, you will feel the pain more and more.

The other part of the problem is unit testing. When an application is tightly coupled, you can only do functional (or user interface) testing of the application. Again, this is not an issue with a small project, but as a project grows in size and complexity, being able to test application layers separately becomes very important. Keep in mind that unit testing is not just about making sure that a unit works when you use it in a system but about making sure it continues to work in a system. Having unit tests for parts of a system adds assurance that as the system changes, problems are revealed earlier in the process rather than later (as would happen with functional testing). Regression testing (for example, running unit tests on a system

on every build) then becomes crucial to ensuring that small changes that are added to a system are not going to cause cascading bugs.
Creating an application by defining different layers might appear to some developers to be a case of over-engineering. The fact is that whether you build with layers in mind or not, you are working on an n-tier platform and your application will have layers. But without formal planning, you will end up with either a very tightly coupled system (and the problems detailed previously) or an application full of spaghetti code that will be a maintenance headache.
It is easy to assume that building an application with separate layers or tiers requires a lot of infrastructure to make it work well, but in fact, simple separation of layers is straightforward to implement. (You can design more complex layering of an application by using inversion of control techniques, but that addresses a different problem than is discussed in this article.)

Application Layering in Silverlight 2

Silverlight 2 does not require you to invent something new to help you decide how to layer an application. There are some well-known patterns that you can use for your design.
A pattern that people hear a lot about right now is the Model-View-Controller (MVC) pattern. In the MVC pattern, the model is the data, the view is the user interface, and the controller is the programmatic interface between the view, the model, and the user input. This pattern, however, does not work well in declarative user interfaces like Windows Presentation Foundation (WPF) or Silverlight because the XAML that these technologies use can define some of the interface between the input and the view (because data binding, triggers, and states can be declared in XAML).
Model-View-Presenter (MVP) is another common pattern for layering applications. In the MVP pattern, the presenter is responsible for setting and managing state for a view. Like MVC, MVP does not quite fit the Silverlight 2 model because the XAML might contain declarative data binding, triggers, and state management. So where does that leave us?
Luckily for Silverlight 2, the WPF community has rallied behind a pattern called Model-View-ViewModel (MVVM). This pattern is an adaptation of the MVC and MVP patterns in which the view model provides a data model and behavior to the view but allows the view to declaratively bind to the view model. The view becomes a mix of XAML and C# (as Silverlight 2 controls), the model represents the data available to the application, and the view model prepares the model in order to bind it to the view.
The model is especially important because it wraps the access to the data, whether access is through a set of Web services, an ADO.NET Data Service, or some other form of data retrieval. The model is separated from the view model so that the view's data (the view model) can be tested in isolation from the actual data.
Figure 1 shows an example of the MVVM pattern.

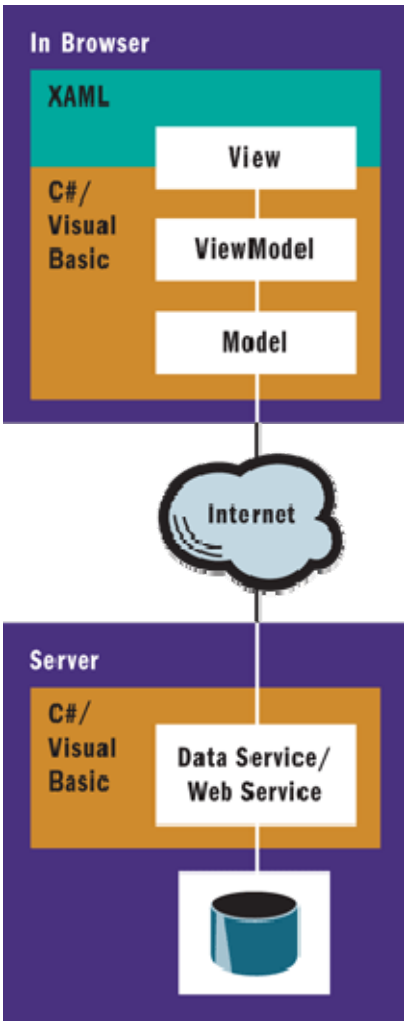

Figure 1 **Model-View-ViewModel Pattern**

MVVM: A Walk-Through

To help you understand how to implement the MVVM pattern, let's walk through an example. This example does not necessarily represent how actual code would be used. It is simply designed to explain the pattern. This example is composed of five separate projects in a single Visual Studio solution. (Although you do not need to create each of the layers as a separate project, it is often a good idea.) The example further separates the projects by placing them in client and server folders. In the Server folder are two projects: an ASP.NET Web Application (MVVMExample) that will host our Silverlight projects and services and a .NET Library project that contains the data model.

In the Client folder are three projects: a Silverlight project (MVVM.Client) for the main UI of our app, a Silverlight client library (MVVM.Client.Data) containing the model and view model as well as service references, and a Silverlight project (MVVM.Client.Tests) containing the unit tests. You can see the breakdown of these projects in **Figure 2**.

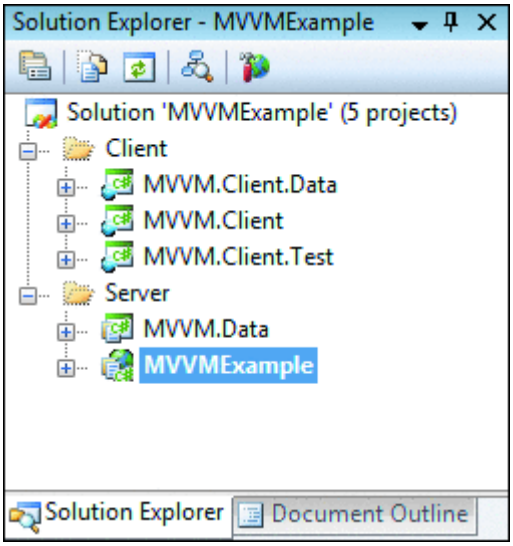

Figure 2 **Project Layout**

For this example, I used ASP.NET, Entity Framework, and an ADO.NET Data Service on the server. Essentially, I have a simple data model on the server that I expose through a REST-based service. Please see my September 2008 article on using ADO.NET Data Services in Silverlight 2, "Data Services: Create Data-Centric Web Applications with Silverlight 2" for a further explanation of those details.

Creating the Model

To enable layering in our Silverlight app, we first need to define the model for the application data in the MVVM.Client.Data project. Part of defining the model is determining the types of entities you are going to work with inside an application. The types of entities depend on how the application will interact with the server data. For example, if you are using Web services, your entities are likely going to be data contract classes that are generated by creating a service reference to your Web service. Alternatively, you could use simple XML Elements if you are retrieving raw XML in your data access. Here , I use an ADO.NET Data Service, so when I create a service reference to the data service, a set of entities is created for me.
In this example, the service reference created three classes that we care about: Game, Supplier, and GameEntities (the context object to access the data service). The Game and Supplier classes are the actual entities that we use to interact with the view, and the GameEntities class is used internally to access the data service to retrieve data.
Before we can create the model, however, we need to create an interface for communication between the model and the view model. This interface typically includes any methods, properties, and events that are required to access data. This set of functionality is represented by an interface to allow it to be replaced by other implementations as necessary (testing, for example). The model interface in this example, shown here, is called IGameCatalog.

Copy Code

```
public interface IGameCatalog
{
  void GetGames();
  void GetGamesByGenre(string genre);
  void SaveChanges();

  event EventHandler<GameLoadingEventArgs> GameLoadingComplete;
  event EventHandler<GameCatalogErrorEventArgs> GameLoadingError;
  event EventHandler GameSavingComplete;
  event EventHandler<GameCatalogErrorEventArgs> GameSavingError;
}
```

The IGameCatalog interface contains methods to retrieve and save data. However, none of the operations return actual data. Instead they have corresponding events for success and failure. This behavior enables asynchronous execution to address the Silverlight 2 requirement for asynchronous network activity. While an

asynchronous design in WPF is often recommended, this particular design works well in Silverlight 2 because Silverlight 2 requires asynchronicity.
To enable notification of the results to the caller of our interface, the example implements a GameLoadingEventArgs class that is used in the events to send the results of a request. This class exposes our entity type (Game) as an enumerable list of entities that contains the results the caller requested, as you can see in the following code.

Copy Code

```
public class GameLoadingEventArgs : EventArgs
{
  public IEnumerable<Game> Results { get; private set; }

  public GameLoadingEventArgs(IEnumerable<Game> results)
  {
    Results = results;
  }
}
```

Now that we have defined our interface, we can create the model class (GameCatalog) that implements the IGameCatalog interface. The GameCatalog class simply wraps the ADO.NET Data Service so that when a request for data comes in (GetGames or GetGamesByGenre), it executes the request and throws an event that contains the data (or an error, if one occurs). This code is meant to simplify access to the data without imparting any specific knowledge to the caller. The class includes an overloaded constructor to specify the URI of the service, but that is not always needed and could be implemented as a configuration element instead. **Figure 3** shows the code for the GameCatalog class.

Figure 3 The GameCatalog Class

Copy Code

```
public class GameCatalog : IGameCatalog
{
  Uri theServiceRoot;
  GamesEntities theEntities;
  const int MAX_RESULTS = 50;

  public GameCatalog() : this(new Uri("/Games.svc", UriKind.Relative))
  {
  }

  public GameCatalog(Uri serviceRoot)
  {
    theServiceRoot = serviceRoot;
  }

  public event EventHandler<GameLoadingEventArgs> GameLoadingComplete;
  public event EventHandler<GameCatalogErrorEventArgs> GameLoadingError;
  public event EventHandler GameSavingComplete;
  public event EventHandler<GameCatalogErrorEventArgs> GameSavingError;

  public void GetGames()
  {
    // Get all the games ordered by release date
    var qry = (from g in Entities.Games
               orderby g.ReleaseDate descending
               select g).Take(MAX_RESULTS) as DataServiceQuery<Game>;

    ExecuteGameQuery(qry);
  }

  public void GetGamesByGenre(string genre)
```

```
  {
    // Get all the games ordered by release date
    var qry = (from g in Entities.Games
               where g.Genre.ToLower() == genre.ToLower()
               orderby g.ReleaseDate
               select g).Take(MAX_RESULTS) as DataServiceQuery<Game>;

    ExecuteGameQuery(qry);
  }

  public void SaveChanges()
  {
    // Save Not Yet Implemented
    throw new NotImplementedException();
  }

  // Call the query asynchronously and add the results to the collection
  void ExecuteGameQuery(DataServiceQuery<Game> qry)
  {
    // Execute the query
    qry.BeginExecute(new AsyncCallback(a =>
    {
      try
      {
        IEnumerable<Game> results = qry.EndExecute(a);

        if (GameLoadingComplete != null)
        {
          GameLoadingComplete(this, new GameLoadingEventArgs(results));
        }
      }
      catch (Exception ex)
      {
        if (GameLoadingError != null)
        {
          GameLoadingError(this, new GameCatalogErrorEventArgs(ex));
        }
      }

    }), null);
  }

  GamesEntities Entities
  {
    get
    {
      if (theEntities == null)
      {
        theEntities = new GamesEntities(theServiceRoot);
      }
      return theEntities;
    }
  }
}
```

Notice the ExecuteGameQuery method, which takes the ADO.NET Data Service query and executes it. This method executes the result asynchronously and returns the result to the caller.

Note that the model executes the query but simply fires events when it is complete. You might look at this and wonder why the model doesn't ensure that the events marshal the calls to the user interface thread in Silverlight 2. The reason is that Silverlight (like its other user interface brethren, such as Windows Forms and WPF) can only update the user interface from a main or UI thread. But if we do that marshaling in this code, it would tie our model to the user interface, which is exactly counter to our stated purpose (separating the concerns). If you assume that the data needs to be returned on the UI thread, you bind this class to user interface calls, but this is antithetical to why you use separate layers in an application.

Views and the View Model

It might seem obvious to create the view model to expose data directly to our view class(es). The problem with this approach is that the view model should only expose data that is directly needed by the view; therefore, you need to understand what the view needs. In many cases, you will be creating the view model and the view in parallel, refactoring the view model when the view has new requirements. Although the view model is exposing data to the view, the view is also interacting with the entity classes (indirectly because the model's entities are being passed to the view by the view model).

In this example, we have a simple design that is used to browse XBox 360 game data, as shown in **Figure 4**. This design implies that we need a list of Game entities from our model filtered by genre (selected via the drop-down list). To fulfill this requirement, the example needs a view model that exposes the following:

- A data-bindable Game list for the currently selected genre.
- A method to make the request for the genre selected.
- An event that alerts the UI that the list of games has been updated (because our data requests will be asynchronous).

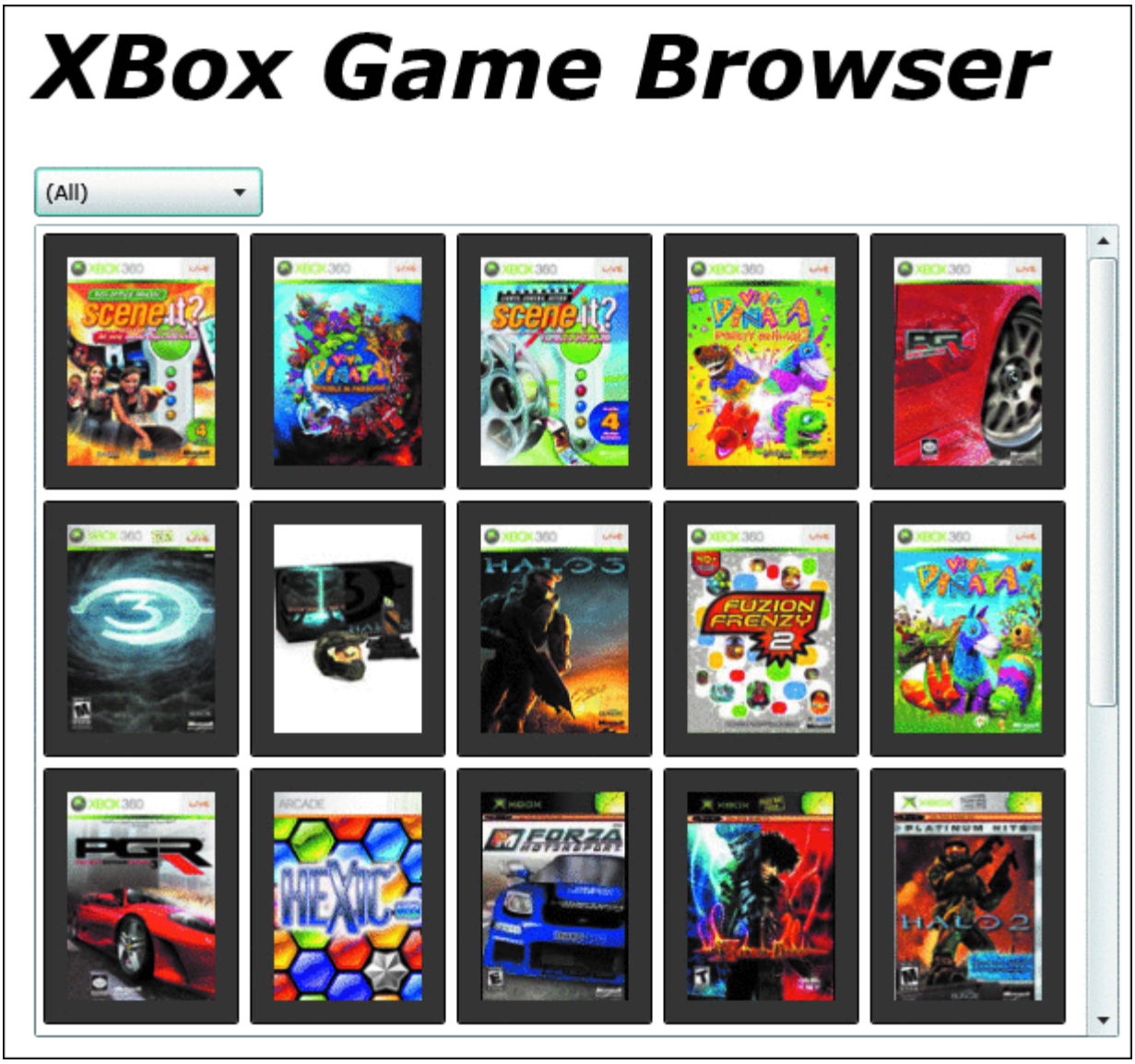

Figure 4 **Example User Interface**

Once our view model supports this set of requirements, it can be bound to the XAML directly, as shown in GameView.XAML (located in the MVVM.Client project). This binding is implemented by creating a new instance of the view model in the Resources of the view and then binding the main container (a Grid in this case) to the view model. This implies that the entire XAML file will be data bound based on the view model directly. **Figure 5** shows the GameView.XAML code.

Figure 5 GameView.XAML

Copy Code

```
// GameView.XAML
<UserControl x:Class="MVVM.Client.Views.GameView"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:data="clr-
namespace:MVVM.Client.Data;assembly=MVVM.Client.Data">

  <UserControl.Resources>
    <data:GamesViewModel x:Key="TheViewModel" />
  </UserControl.Resources>
```

```
  <Grid x:Name="LayoutRoot"
        DataContext="{Binding Path=Games, Source={StaticResource TheViewModel}}">
    ...
  </Grid>
</UserControl>
```

Our view model needs to fulfill these requirements using an IGameCatalog interface. In general, its useful to have the default constructor of a view model create a default model so that binding to the XAML is easy, but you should also include an overload of the constructor in which the model is supplied to allow for scenarios such as testing. The example view model (GameViewModel) looks like **Figure 6**.

Figure 6 GameViewModel Class

Copy Code

```
public class GamesViewModel
{
  IGameCatalog theCatalog;
  ObservableCollection<Game> theGames = new ObservableCollection<Game>();

  public event EventHandler LoadComplete;
  public event EventHandler ErrorLoading;

 public GamesViewModel() :
    this(new GameCatalog())
  {
  }

  public GamesViewModel(IGameCatalog catalog)
  {
    theCatalog = catalog;
    theCatalog.GameLoadingComplete +=
      new EventHandler<GameLoadingEventArgs>(games_GameLoadingComplete);
    theCatalog.GameLoadingError +=
      new EventHandler<GameCatalogErrorEventArgs>(games_GameLoadingError);
  }

  void games_GameLoadingError(object sender, GameCatalogErrorEventArgs e)
  {
    // Fire Event on UI Thread
    Application.Current.RootVisual.Dispatcher.BeginInvoke(() =>
      {
        if (ErrorLoading != null) ErrorLoading(this, null);
      });
  }

  void games_GameLoadingComplete(object sender, GameLoadingEventArgs e)
  {
    // Fire Event on UI Thread
    Application.Current.RootVisual.Dispatcher.BeginInvoke(() =>
      {
        // Clear the list
        theGames.Clear();

        // Add the new games
        foreach (Game g in e.Results) theGames.Add(g);

        if (LoadComplete != null) LoadComplete(this, null);
      });
  }
```

```
  public void LoadGames()
  {
    theCatalog.GetGames();
  }

  public void LoadGamesByGenre(string genre)
  {
    theCatalog.GetGamesByGenre(genre);
  }

  public ObservableCollection<Game> Games
  {
    get
    {
      return theGames;
    }
  }
}
```

Of particular interest in the view model are the handlers for GameLoadingComplete (and GameLoadingError). These handlers receive events from the model and then fire events to the view. What is interesting here is that the model passes the view model the list of results, but instead of passing the results directly to the underlying view, the view model stores the results in their own bindable list (ObservableCollection<Game>).

This behavior occurs because the view model is being bound directly to the view, so the results will show up in the view via the data binding. Because the view model has knowledge of the user interface (because its purpose is to fulfill UI requests), it can then make sure that the events that it fires happen on the UI thread (via Dispatcher.BeginInvoke, although you can use other methods for calling on the UI thread if you prefer).

Concessions to Silverlight 2

The MVVM pattern is used throughout many WPF projects to great success. The problem with using it in Silverlight 2 is that to make this pattern easy and seamless, Silverlight 2 really needs to support Commands and Triggers. If that were the case, we could have the XAML call the methods of the view model directly when the user interacts with the application.

In Silverlight 2 this behavior requires a bit more work, but luckily it involves writing only a little code. For example, when the user selects a different genre using the drop-down list, we would like to have a Command that executes the GameViewModel.GetGameByGenre method for us. Because the infrastructure required is not available, we simply have to use code to do the same thing. When the combo box's (genreComboBox) selection changes, the example loads the Games from the view model manually in code instead of in a Command. All that is required here is that the request to load the data happens—because we are bound to the list of Games, the underlying view model will simply change the collection we are bound to and the updated data appears automatically. You can see this code in **Figure 7**.

Figure 7 Updating Data in the UI

Copy Code

```
void genreComboBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
  string item = genreComboBox.SelectedItem as string;
  if (item != null)
  {
    LoadGames(item);
  }
}

void LoadGames(string genre)
{
  loadingBar.Visibility = Visibility.Visible;
```

```
  if (genre == "(All)")
  {
    viewModel.LoadGames();
  }
  else
  {
    viewModel.LoadGamesByGenre(genre);
  }

}
```

There are several places where the lack of element binding and commands will force Silverlight 2 developers to handle this behavior in code. Because the code is part of the view, this does not break the layering of the application, it is just not as straightforward as the all-XAML examples you will see in WPF.

Where We Are

Silverlight 2 does not require you to build monolithic applications. Layering Silverlight 2 applications is straightforward using the Model-View-ViewModel pattern that we happily borrow from our WPF brethren. Furthermore, using this layering approach allows you to loosely couple the responsibilities in your applications so that they are easier to maintain, extend, test, and deploy.

I'd like to thank the Laurent Bugnion (author of Silverlight 2 Unleashed) as well as others on the WPF Disciples mailing list for their help in completing this article. Laurent blogs at blog.galasoft.ch.

Shawn Wildermuth is a Microsoft MVP (C#) and the founder of Wildermuth Consulting Services. He is the author of several books and numerous articles. In addition, Shawn currently runs the Silverlight Tour teaching Silverlight 2 around the country. He can be contacted at shawn@wildermuthconsulting.com.