User interface design

in C#, with Windows Presentation Foundation (WPF)

# Lab 4 – Using the Model-View-ViewModel Pattern

## Functional Goals

Architect the MyTVCompanion app using the MVVM pattern to allow for better scalability and separation of business logic and UI concerns.

## Learning Goals

* Understand the Model-View-ViewModel pattern
* Understand DataContexts

## Prerequisites

* You’ll need to install Visual Studio (2012 Ultimate was used to create this lab) from the MSDN/DreamSpark service on ANGEL’s RosePortal.
* Code from Lab3 (a complete version may be available from your instructor)

## Submission Instructions

Submit answers to the **3** questions in this lab as a .pdf to the appropriate Moodle submission form.

## Getting started: The MVVM pattern

Open up Visual Studio and look at the code in MainWindow.xaml.cs, SettingsWindow.xaml.cs, and App.xaml.cs. It’s kind of a mess – there’s UI code and model-level (i.e. TVDB interaction) code all mixed in together. Right now that’s alright, because this is a small project; but, what if you keep adding features? What if the TVDB library you’re using right now ceases to work, or you lose the rights to use it? You’re going to need to refactor almost all of your code to make the project work again.

Enter Model-View-ViewModel, or MVVM. MVVM is the WPF incarnation of Model-View-Controller. For some background on MVVM, head to http://msdn.microsoft.com/en-us/magazine/dd419663.aspx#id0090009 and read “The Evolution of Model-View-ViewModel” and “Why WPF Developers Love MVVM”. You’re welcome to read the whole article, but it goes farther than this lab will go and uses techniques you may not be familiar with yet.

Question 1: What are 3 advantages of MVVM according to the MSDN article linked above? (6 points)

Now that you know a little bit about MVVM, you’re going to convert your MyTVCompanion application to follow the MVVM pattern.

## Creating and referencing a ViewModel

1. Right-click on the MyTVCompanion project and select Add -> New Folder. Call the folder ViewModel.
2. Right-click on the ViewModel folder you created and select Add -> Class. Call the class MainWindowViewModel.
3. In MainWindow.xaml.cs, right after the class opening curly brace, use the dependencyProperty snippet to insert a new property. Tab your way through the snippet.
   1. The first tab stop is the name of the property – type ViewModel here and press tab.
   2. The second tab stop is the type the property refers to – type MainWindowViewModel here and press tab.
   3. The third tab stop is the type of the object containing the property – leave this as is and press tab.
4. In MainWindow.xaml, edit the Window DataContext:

DataContext="{Binding ViewModel, RelativeSource={RelativeSource Self}}"

## Separating your concerns

Now that you have a ViewModel, your next step is to move all model code (i.e. all code that deals with TV shows or isolated storage) into the ViewModel. When you’re done, the only code left in MainWindow.xaml.cs should be a reference to the ViewModel and event handler work that directly relates to the view (but is independent of the model).

### Moving isolated storage connections

1. Open App.xaml.cs and MainWindowViewModel.cs side-by-side.
2. Refactor the code in the constructor that gets the shows from IsolatedStorage into a new method called GetShowsFromIsolatedStorage.
3. Move your new method, AppExit, and the two related fields into MainWindowViewModel.cs.
   1. Rename AppExit to Exit and remove its parameters: public void Exit().
   2. Add a new call to GetShowsFromIsolatedStorage to the constructor for MainWindowViewModel (you may need to create the constructor – use the ctor snippet).
4. App.xaml.cs should now have only two properties and one line of constructor code dealing with TvdbHandler.
5. Open App.xaml and remove the Exit line from the Application tag.

Application.Current.Exit += (sender, args) => ViewModel.Exit();

1. Open MainWindow.xaml.cs; in the constructor, add this line:

### Moving Shows and TVDB references

You’ve now seen how to refactor code into a ViewModel and access it from the codebehind. For the rest of this lab, replace the contents of your MainWindowViewModel.cs and MainWindow.xaml.cs with the code on the last two pages of the lab. Examine these files and observe how your code now interacts with the ViewModel. Notice that MainWindow.xaml.cs doesn’t have any direct knowledge of the TVDB library – all it cares about is the GetDayEpisodes method, which hides all of the actual TVDB communication.

## Your turn: Settings window

Refactor the SettingsWindow.xaml.cs codebehind in the same style. When you’re finished, SettingsWindow should have no knowledge of the TVDB library or its methods, save for the ObservableCollection<TvdbSeries> given to the constructor by MainWindow.xaml.cs. Hint: Your SettingsWindowViewModel will need both things passed to that constructor (new SettingsWindow(ViewModel.TvdbHandler, ViewModel.Shows);).

When you’ve finished, run your application – it should work just as it did before.

Question 2: Submit the C# code for SettingsWindow.xaml.cs and SettingsWindowViewModel.cs. If you are using a word processor that doesn’t retain Visual Studio’s text formatting on copy/paste, please take a screenshot of your code so that it remains properly formatted and colored. (24 points)

## Some caveats

Now that you’re done, you’ve achieved a much better separation of concerns using the MVVM pattern. However, we didn’t go all the way – SettingsWindow still knows about the TvdbSeries insofar as it has to pass a reference to the list of shows from one ViewModel to the next. This leads to a general rule of MVVM – if it’s easier to do something directly in the codebehind, but that something breaks your separation of concerns, it **might** be ok to do it in the codebehind. In this case, modifiability isn’t terribly affected – a future version of this application would only have to change one constructor in the View if the underlying libraries changed.

Bottom line: if something’s easy in codebehind but really hard in MVVM, do it in the codebehind; just be aware of the modifiability tradeoff you’re making.

Question 3: How might you alter the SettingsWindow constructor so that it doesn’t have to know about the TvdbSeries class? (3 points)

## Congratulations and a warning

You’re done! Don’t forget: submit answers to the **3** questions in this lab as a .pdf to the appropriate Moodle submission form.

A final note: MVVM is a little weird until you get used to it. WPF does a lot for you behind the scenes, which is great when it works right, but it can be frustrating to sort out when it doesn’t. Stick with it, though, and your code will become easier to write and have an excellent separation of concerns. Also remember that MVVM is a widely-used pattern without a universal set of best practices (see http://stackoverflow.com/questions/291518/inotifypropertychanged-vs-dependencyproperty-in-viewmodel for one of the larger debates in the MVVM community), so there will be points where you just need to do what works best for you in a particular situation.

## MainWindowViewModel.cs

Replace everything but the two isolated storage methods you created earlier with this code.

private IsolatedStorageFile \_isolatedStorage;

private const String ShowsFileName = "mydata.bin";

internal TvdbHandler TvdbHandler { get; private set; }

public ObservableCollection<TvdbSeries> Shows { get; private set; }

public ObservableCollection<String> SelectedDayEpisodes { get; private set; }

public MainWindowViewModel()

{

TvdbHandler = new TvdbHandler("49FF3082EF06CF50");

GetShowsFromIsolatedStorage();

SelectedDayEpisodes = new ObservableCollection<String>();

GetDayEpisodes(DateTime.Today);

}

public void GetDayEpisodes(DateTime day)

{

SelectedDayEpisodes.Clear();

var seriesThatAirToday = Shows.Where(series => series.AirsDayOfWeek == day.DayOfWeek).ToList();

var episodesToSort = (from series in seriesThatAirToday

let episode =

series.GetEpisodes(series.NumSeasons).FindLast((ep) => ep.FirstAired == day.Date)

where episode != default(TvdbEpisode)

select new EpisodeToSort() { Episode = episode, Series = series })

.ToList();

episodesToSort.Sort((c1, c2) =>

{

var c1Time = DateTime.Parse(c1.Series.AirsTime);

var c2Time = DateTime.Parse(c2.Series.AirsTime);

if (c1Time < c2Time) return -1;

if (c1Time > c2Time) return 1;

return 0;

});

foreach (var e in episodesToSort)

SelectedDayEpisodes.Add(e.Series.AirsTime + " - " + e.Series.SeriesName + ": " + e.Episode.EpisodeName);

}

private struct EpisodeToSort

{

public TvdbSeries Series;

public TvdbEpisode Episode;

}

## MainWindow.xaml.cs

public MainWindow()

{

ViewModel = new MainWindowViewModel();

InitializeComponent();

Calendar.SelectedDate = DateTime.Today;

Application.Current.Exit += (sender, args) => ViewModel.Exit();

}

private void OnSettingsButtonClick(object sender, RoutedEventArgs e)

{

var window = new SettingsWindow(ViewModel.TvdbHandler, ViewModel.Shows);

window.Closed += (o, k) =>

ViewModel.GetDayEpisodes(Calendar.SelectedDate.GetValueOrDefault());

window.Owner = this;

window.Show();

}

private void CalendarSelectedDatesChanged(object sender, System.Windows.Controls.SelectionChangedEventArgs e)

{

ViewModel.GetDayEpisodes((DateTime)e.AddedItems[0]);

}

Replace everything but the ViewModel property you created earlier with this code.