AARHUS UNIVERSITY SCHOOL OF ENGINEERING

Electro and Information Communication Technology Department

The Model-View-ViewModel Design Pattern

Poul Ejnar Rovsing

Version 0.51 – January 2014

Preface

You can't find much new information regarding the Model-View-ViewModel (MVVM) pattern in this writing – all you need to know can be found on Internet. The reason for this note is the lack of one suitable source on the Internet that can guide newbies through a describing introduction, over the basic implementation details to all the advanced scenarios related to the use of the MVVM pattern - and have a primary focus on WPF.

The reader is expected to have a basic knowledge of WPF and data binding to fully comprehend this writing.

Source code for all examples can be downloaded from Campusnet – I4SWD - Filesharing – Execises – 07 MVP and MVVM.

Table of Contents

Chapter

1

1 Introduction

Although there are many suitable design patterns for programming graphical user interfaces almost all WPF developers uses the MVVM pattern. In this chapter I will try to explain why this is so and where this pattern came from.

You don't have to use a design pattern at all. If the application you are going to develop is very small and you don't expect it to be around for long, then the development process will probably not gain much from use of design patterns – unless you are familiar with the MVVM pattern and just can't live without it. But a project doesn't need to be big to benefit from the use of the MVVM pattern. Even single-person projects can benefit from a more testable and maintainable code base that can evolve over time using the modular approach.

1.1 Where it came from

John Grossman, one of the WPF architects at Microsoft, presented the MVVM pattern on his blog: "Tales from the Smart Client" to world in 2005 (Gossman 2005). Grossman writes:

> Model/View/ViewModel is a variation of Model/View/Controller (MVC) that is tailored for modern UI development platforms where the View is the responsibility of a designer rather than a classic developer. The designer is generally a more graphical, artistic focused person, and does less classic coding than a traditional developer. The design is almost always done in a declarative form like HTML or XAML, and very often using a WYSIWYG tool such as Dreamweaver, Flash or Sparkle. In short, the UI part of the application is being developed using different tools, languages and by a different person than is the business logic or data backend. Model/View/ViewModel is thus a refinement of MVC that evolves it from its Smalltalk origins where the entire application was built using one environment and language, into the very familiar modern environment of Web and now Avalon development.
>
> Model/View/ViewModel also relies on one more thing: a general mechanism for data binding. (Gossman 2005)

To fully understand Grossmans' text you should know that Sparkle was the code name for Expression Blend and Avalon was the code name for WPF. Grossman had developed the MVVM pattern to make it easier for the UI designer to work independently from the developers. And the main technique for this separation is data binding.

1.2 Why use MVVM?

There are several reasons to use the MVVM pattern:

Separation of concerns
Grossman clearly states that the main motivation for use of the MVVM pattern is separation of concerns. The UI designer responsible for the view is free to use other languages and tools than the developers responsible for the business logic and data backend (Gossman 2005). And the developers have two (or more) main places to put their work. Presentation logic goes into the ViewModel and business logic goes into the model, and data access code into the DAL layer.

Testability
The ViewModel has no dependencies on the View which make it easy to substitute the view with unit tests. And if the design of a MVVM application is done right there are no GUI controls (widgets) in the ViewModel and no user intervention (like a messageBox that pops up), so the automated unit tests can run unattended. A clever developer can often place all presentation logic in the ViewModel and no code in the View's code-behind file, which make it possible to get 100% code coverage in unit tests – if you omit the Xaml-code from the calculation. For many developers this is the main argument for using MVVM, but for many projects the other arguments can be equally important – if not more.

Low coupling
The MVVM pattern set the stage for an application with very low coupling between the classes that make up the application. And if you add proper use of interfaces, a message bus (aka. Event Aggregator) and use of locator pattern, factory pattern or a dependency injection framework (IOC container) then you can get a modular application that can evolve over the years to come.

A smooth designer/developer workflow
There is a very loose connection between a View and its ViewModel, so a designer can work on the View at the same time as a developer works on the ViewModel.

Portability
The MVVM pattern is not restricted to WPF applications for the Windows desktop. You can also use the MVVM pattern in Windows Store applications, in Windows Phone applications, Silverlight applications and in applications for the Xbox. The View has to be built specific to the platform you are targeting, but the ViewModel and Model can be built to run on all platforms if you put them in a Portable Class Library.

1.3 The MVC and MVP Patterns

The ancestors to the MVVM pattern are the MVC and MVP patterns, so I will present them before I dig into the MVVM pattern in the next chapter. If you are already familiar with these patterns you may skip this section.

1.3.1 Model View Controller (MVC)

Origin

The Model View Controller design pattern (MVC) was developed by the Norwegian professor Trygve Reenskaug during his stay at Xerox Parc in 1978 – 1979. The group Reenskaug worked with experimented with development of graphical user interfaces and MVC pattern was conceived as a general solution to the problem of users controlling a large and complex data set (Reenskaug 2003) and the capability to have several views visualizing the same data set.

Jim Althoff and others implemented a version of MVC for the Smalltalk-80 class library after Reenskaug had left Xerox Parc, and they used the term Controller somewhat differently from Reenskaug's original idea (Reenskaug 2003). A description of this implementation can be found in the work of S. Burbeck (Burbeck 1992) and this is the version I will present here.

A more thorough description of the MVC pattern can be found in book Pattern-Oriented Software Architecture 1 page 125 (Buschmann, Meunier et al. 1996)

Structure

Figure 1 shows the structure of Model-View-Controller pattern as implemented in the Smalltalk-80 class library. This figure is a simplistic visualization of the principal relationships in the MVC pattern. In a real application there will typical be many different View classes, Controller classes and Model classes.

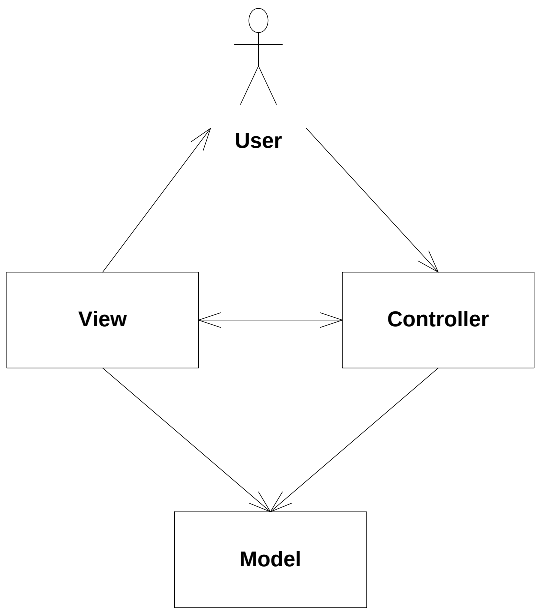

Figure 1: The structure of the MVC pattern

Model

The model holds the data when the application is running, and also contains the business logic. How the model is organized internally is free for the developer to decide. But it is mandatory that the model does not depend on the View or Controller. If the Model has to update the View or Controller this must be done by use of the Observer Design Pattern (or a similar notification mechanism).

View

The view is responsible for showing the model to the user – or more often: it will show certain parts or properties of the model on the screen (GUI). A View typical only has a minor area of the screen real estate as its output area, but it may own the whole screen. Several Views can be connected to the same model – each View will visualize different properties of the model, or visualize the data in different ways (e.g. as numbers or as a bar chart). The View is dependent on both the model and its associated controller. The View does not send a lot of messages to its Controller, but the View creates and releases its Controller.

Controller

The Controller gets all input from the user such as keyboard stokes and mouse. Depending on the kind of input the Controller gets it will update the model or send commands to its associated View. When the controller updates the model, the View will be notified from the Model through the observer design pattern. In an application there will be one Controller for each View, but only one Controller will receive input at any one time.

Key concepts

A strong separation between the presentation layer (View and Controller) and the business logic layer (Model) which facilitates reuse of the model in new presentation frameworks. This is very important as time has shown that presentation frameworks changes more rapidly than business logic.

Use of the Observer design pattern makes it possible to have multiple Views connected to the same Model and still maintain a loose coupling in the application.

Variations

The MVC pattern in its original form as presented above is seldom used today, but there are many variants of sthe original pattern that people still refer to as the MVC pattern – sometimes quite misleading.

Document-View:

In this variant the View and Controller is collapsed to one class. Most modern GUI frameworks include a lot of generic widgets (called controls on the Windows platform) that both handle output and input and thereby make the original Controller in MVC obsolete.

The Web-based MVC pattern:

Many Web server frameworks use the MVC-pattern, but they typical have the Controller as the main component and responsible for creation of the appropriate View, and they may have a front-controller responsible for handling requests over to the relevant controller (Greer 2007). Sun's Model 2 is the first well known Web server framework that follows the MVC-pattern (Seshadri 1999), but there are several others e.g. ASP.Net MVC.

1.3.2 Model View Presenter (MVP)

Origin

The Model View Presenter design pattern (MVP) was developed by Mike Potel and his colleges at Talingnt Inc. for use in their the Open Class class libraries for IBM's VisualAge programming environments (Potel 1996) during the ears 1992 – 1996. The evolution of Operating Systems like Microsoft's Windows had resulted in generic GUI widgets (controls) that contained both representation and initial user input handling making the original MVC pattern awkward to use. But just collapsing the View and Controller as done in the Document-View variation of MVC can result in huge classes with too many responsibilities and too little cohesion. To solve this they introduced the Presenter which is a transformation of the Controller to fit in the new programming model. The Taligent version of the MVP pattern actually included 3 more components in the pattern: Commands, Selections and Interactors. But I won't describe them here as the other MVP dialects don't include them.

In 1997, Andy Bower and Blair McGlashan of Object Arts Ltd. adapted the MVP pattern to form the basis for their Dolphin Smalltalk user interface framework (Bower, McGlashan 2000). Although their version of the MVP pattern is a little different than original Taligent version, it is the Dolphin version I will present here, as I see their version as the one that best represent the fundamentals of the MVP-pattern.

Structure

As seen in Figure 2: The structure of the MVP pattern is similar to the MVC pattern. The only difference is that is no direct connection from the user to the Controller. All the communication with the user is through the View.

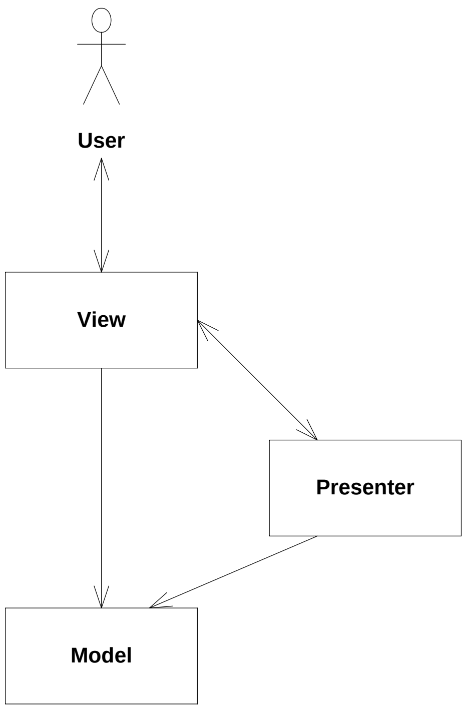

Figure 2: The structure of the MVP pattern

Model

Like in the MVC pattern the model holds the data while the application is running, and it also contains all the business logic. How the model is organized internally is free for the developer to decide. But it is mandatory that the model does not depend on the View or Presenter. If the Model has to update the View or Presenter this must be done by use of the Observer design pattern (or a similar mechanism).

View

The view is responsible for both showing the model on the screen and the initial handling of user input. Several Views can be connected to the same model – each View will visualize different properties of the model, or visualize the data in different ways (e.g. as numbers or as a bar chart). The View can handle some user input itself, but often it delegates the handling of user input to the Presenter. The View is dependent on both the model and its associated Presenter.

Presenter

The Presenter is responsible for the main part of updates to the model, as the View only handle the simple cases itself. So the Presenter is where most of the applications behaviour is placed.

Key concepts

The MVP pattern has the same prime concepts as the MVC pattern:

- A strong separation between the presentation layer (View and Presenter) and the business logic layer (Model).
- Use of the Observer design pattern makes it possible to have multiple Views connected to the same Model and still maintain a loose coupling in the application.

Differences between MVC and MVP

So where are the fundamental differences between MVC and MVP patterns?

1. In MVP all user interaction goes through the View.
2. In MVC the Controller handle all user input and updating the Model is only a part of the Controllers responsibility.
   In MVP the Presenters main responsibility is to update the Model.
3. In MVP the View typical contains generic widgets supplied by the operating system or GUI framework.

Variations

There are many variations in the way developers implement the MVP pattern. What differs most is how the View is updated. Martin Fowler has described and named the popular variants (Fowler 2006) I will mention here. Be aware that the Presenter is named Controller in some variations.

Passive View (Fowler 2006):

In this variation the View has no direct association to the Model. The View only contains the widgets that shows the data on the screen, and it will hand over all user input to the Presenter (Fowler call it Controller). The Presenter contains all the GUI logic and is responsible for both updating the Model and updating the View – the Presenter populates the Widgets in the View with data from the Model. This variation of MVP is the one that Wikipedia presents as the MVP pattern.

Presentation Model (Fowler 2004):
As in the Passive View variation the View has no direct association to the Model but only to the Presentation Model (a new name for the Presenter). But in this variation the View itself is responsible for populating the widgets with data from the Presentation Model. The Presentation Models responsibility is to make it easy for View to map to the relevant data in the Presentation Model and the Presentation Model is also responsible for updates to the Model.

Supervising Controller (Fowler 2006):
This is the name Fowler gives the Dolphin version of the MVP pattern. The View use data binding or observer synchronization to fetch data from the Model where possible and the more complex relations are handled by the Presenter (or Controller).

Chapter

2

2 The MVVM Pattern Basics

Here you will see the fundamental basics of the Model-View-ViewModel pattern without use of a framework. In a later chapter you can read about some of the many MVVM frameworks that make your life as an application developer easier.

As you may remember from chapter 1 the crater of the Model-View-ViewModel (MVVM) pattern John Grossman presented the MVVM pattern as a variation of the Model-View-Controller pattern (Gossman 2005). But according to Josh Smith the MVVM pattern is better described as a specialization of Fowlers Presentation Model Pattern (Smith 2009), and thereby a descendent of the famous MVP pattern.

2.1 Structure

The View is connected to the ViewModel through data binding and sends commands to the View-Model.

- The ViewModel is unaware of the View.

The ViewModel may interact with Model through properties, method calls and may receive events from the model.

- The Model is unaware of the ViewModel.

The class diagram of the MVVM pattern as show in Figure 3 shows that the basic structure of the MVVM pattern is the same as Fowler's Presentation Model. The main topic that differentiates MVVM from Presentation Model is the pervasive use of declarative data binding in the View.

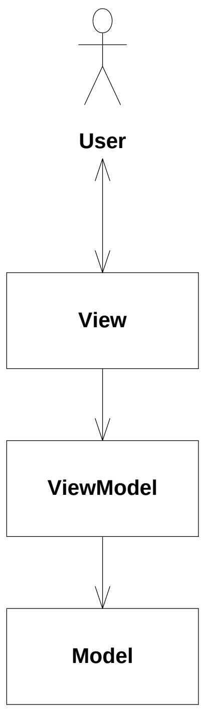

Figure 3: Class diagram of the MVVM pattern

You should be aware that the MVVM pattern is not always implemented as show Figure 3, because by use of Data Binding the View does not need an association to the ViewModel – it can retrieve the ViewModel by use of a Locator or by dependency injection from an IoC-container and then the structure will be as shown in Figure 4 – more one these topics later.

The View is always related to one ViewModel. A ViewModel may serve more than one View, but this is not the typical case. On the class diagrams in Figure 3 and Figure 4 the relation from the ViewModel to the Model is shown as a one to one association, but a ViewModel may have many associations to different classes in the Domain Layer.

2.1.1 VIEW

The view is responsible for both showing the model on the screen and the initial handling of user input. The View has a relation to a ViewModel that holds the state of the View. This relation may be implemented as an association to the ViewModel as shown in Figure 3, but very often it gets the ViewModel from a Locator or by use of dependency injection in some form and the structure will then look like Figure 4. The communication between a View and its ViewModel is mainly done by use of Data Bindings where the View's DataContext property holds the reference to the ViewModel.

The View may be a Window but it could instead be a User Control or a Data Template. It is often used to structure the GUI as a Window containing several Views – where each View is implemented as a User Control or Data Template.

2.1.2 VIEWMODEL

The ViewModel holds the View's state and contains the presentation logic - it acts as a middle man between the View and the Model.

According to the creator of the MVVM pattern the term ViewModel means "Model of a View", and can be thought of as abstraction of the View (Gossman 2005). But Grossman also states that the ViewModel provides a specialization of the Model that the View can use for data binding and this description of the ViewModel is the one that I find best describe the ViewModel's purpose in the MVVM pattern.

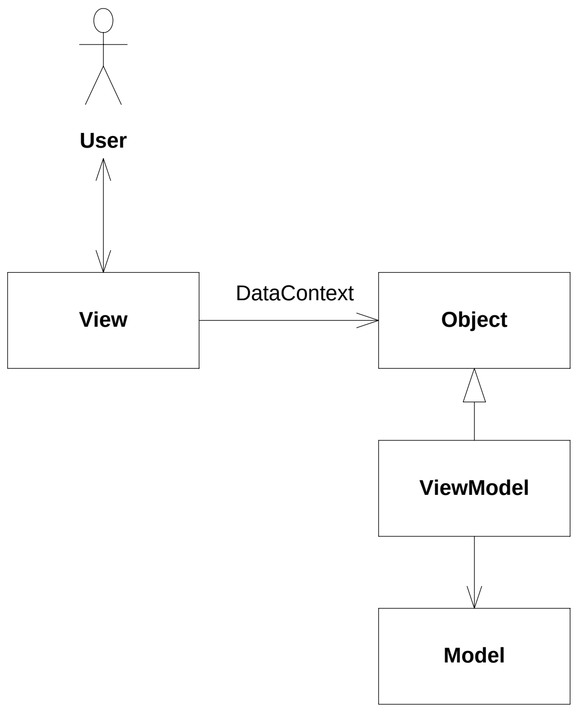

Figure 4: the MVVM pattern with no ViewModel association.

Where possible the View may bind directly to data in the Model, but often this is not possible, and the ViewModel is then used to convert the data from the Model to a format suitable for the View to bind to. The ViewModel also contains Commands the View can use to interact with the Model.

2.1.3 Model

Like in the MVC and MVP patterns the Model holds the data while the application is running and it also contains all the business logic. It is mandatory that the Model is unaware of the used GUI framework (e.g. WPF) and the applications classes in the presentation layer: the View and the ViewModel. These requirements make it easy to reuse the Model in other frameworks.

In real applications the Model is typical a package (or layer) containing several classes. But in some applications there is no Model at all. For applications that just show data stored in a database and where there is very little business logic, the Model is sometimes omitted and the ViewModel will access the persistence layer directly.

2.2 A simple example

To really understand what the MVVM pattern is, you need to see some code. So now it's time to study a very simple WPF application implemented according to the MVVM pattern. As an example application I will use a BMI calculator. When you develop a new application you can either start with the View or with the Model, it doesn't really matter (or if you are in a team you and your team mates can work in parallel). But here I will start with the Model.

My BMIModel looks like this:

```
public class BMIModel
{
    public double Weight { set; get; }

    public double Height { set; get; }

    public double CalculateBMI()
    {
        return Weight / (Height * Height);
    }
}
```

The model is very simple so I won't explain it. The more interesting subject is the ViewModel, which I will present in fragments so I can explain the fundamental parts of the ViewModel.

2.2.1 ViewModel Basics

For data binding to work optimally the ViewModel must implement the `INotifyPropertyChanged` interface (found in the `System.ComponentModel` namespace) and the ViewModel must hold a reference to the Model.

```
public class BMIViewModel : INotifyPropertyChanged
{
        BMIModel bmiModel = new BMIModel();
```

2.2.2 Implementing Properties

Then the ViewModel must expose all the relevant properties for the View to bind to. In this example the ViewModel should expose weight, height and BMI. The Weight and Height properties are implemented quite similar so only the Height property implementation is shown here.

```
public double Height
{
    get { return bmiModel.Height; }
    set
    {
        if (value != bmiModel.Height)
        {
            bmiModel.Height = value;
            NotifyPropertyChanged();
        }
    }
}
```

There are two important issues to remember:

1. You shall only update a property if the new value is different than the old value to avoid unnecessary updates of the user interface.

2. And you must always call the helper function `NotifyPropertyChanged` – even when you know it's not necessary or your code become brittle and may fail when modified later. In this simple example I have implemented the helper function `NotifyPropertyChanged` in the ViewModel, but you would typical place this function in a ViewModelBase class and then inherit this helper function in your concrete ViewModel classes.

```
    public event PropertyChangedEventHandler PropertyChanged;

    private void NotifyPropertyChanged([CallerMemberName] string propertyName
= null)
    {
        var handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
```

In the implementation above there are two implementation details that may need explanation:

1. The attribute `CallerMemberName` in the `System.Runtime.CompilerServices` namespace is a new attribute in C# 5.0 that instructs the compiler to fill in the member name of the calling property if the caller doesn't provide a name explicit.

2. The local variable handler is used to catch the eventhandlers that PropertyChanged may have references to. This is needed as a precaution in a multithreaded application where an eventhandler otherwise might be unsubscribed between the test in the if statement and the invocation.

2.2.3 IMPLEMENTING COMMANDS

In the View there will be a Button that will initiate the calculation of the BMI index. This calculation is implemented as a function in the Model. To connect these two entities the ViewModel uses a Command. The Button has a property named Command of type ICommand that can take a reference to an ICommand implementation and the Button will then invoke the Command when pressed. The build in Command classes in the WPF framework are not well suited to do the job in a MVVM application because they require code in the View's code behind file. So some clever guys at Microsoft have come up with a different ICommand implementation called DelegateCommand. You can find the DelegateCommand class in the Prism library from Microsofts Practices and Patterns group. DelegateCommand is based on John Gossman's original ICommand implementation from Blend. And in the open source community you can find a slightly different ICommand implementation called RelayCommand created by Josh Smith but also used by Laurent Bugnion in his MVVM framework (Bugnion 2009).

Here I chose to use Josh Smith's RelayCommand which can be found in his open source MVVM framework called MVVM Foundation (Smith 2010). I don't use the whole framework – I just include the source code for the RelayCommand class in my project. There are actually two versions of the RelayCommand class: a generic version that takes a command parameter and a non-generic version that does not.

```
public class RelayCommand<T> : ICommand
{
    public RelayCommand(Action<T> execute)
    { …
    public RelayCommand(Action<T> execute, Predicate<T> canExecute)
    { …
}

public class RelayCommand : ICommand
{
    public RelayCommand(Action execute)
    { …
    public RelayCommand(Action execute, Func<bool> canExecute)
    { …
}
```

The execute parameter takes a delegate to the command handler and the canExecute parameter takes a delegate to a function that tells the View if the command should be enabled or not – if omitted the command is always enabled.

To expose a property of type ICommand for the View to bind to the code looks like this:

```
ICommand _calcBMICommand;
public ICommand CalcBMICommand
{
    get { return _calcBMICommand ?? (_calcBMICommand = new
                              RelayCommand(CalcBMI, CalcBMICanExecute)); }
    }

private void CalcBMI()
{
    bmi = bmiModel.CalculateBMI();
    NotifyPropertyChanged("BMI");
}

private bool CalcBMICanExecute()
{
    if (Weight != 0.0 && Height != 0.0)
        return true;
    else
        return false;
}
```

The code above may look a little odd to you because it uses the seldom used ?? operator, but it's actually rather simple. The ICommand reference _calcBMICommand will be initialized to null when the program starts execution. When the CalcBMICommand property´s get function is called the first time the value of _calcBMICommand is null and the ?? operator will return the right operand which will initiate the _calcBMICommand varable to point to a new instance of the RelayCommand class. And this instance will have its execute delegate pointing to CalcBMI and its canExecute delegate will point to CalcBMICanExecute.

CalcBMI is the command handler that is called when the command is invoked. The calculation of the BMI is not done here because this is business logic and is implemented in the Model. So the command handler in the ViewModel asks the Model to calculate BMI and it stores the result in a data member in the ViewModel that is exposed to the View through a property. For the View to update the screen with the new bmi value the command handler calls NotifyPropertyChanged with the name of the property that exposes the bmi value.

CalcBMICanExecute is function that determines whether the command should be enabled or not. In this case the calculation is only relevant when both the Heigth and Weight properties are not zero.

2.2.4 Implementing the View

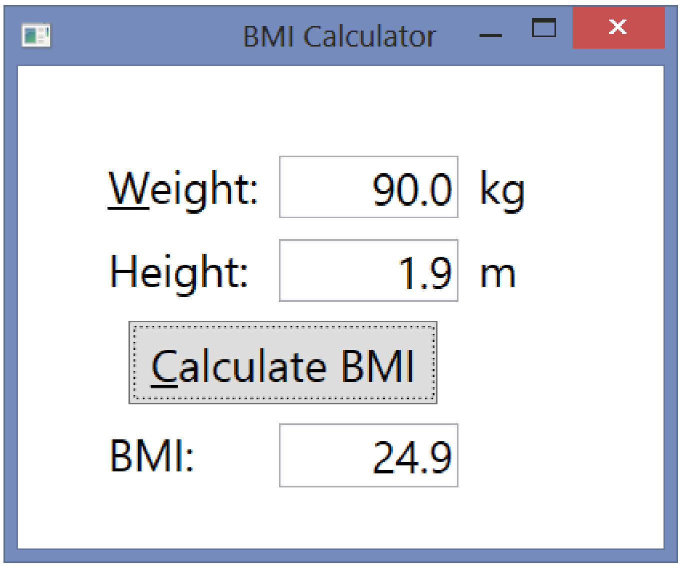

Figure 5: The GUI

As seen in Figure 5 the GUI is quite simple. It is just a bunch of Labels, Textboxes and a single Button. To use the ViewModel it must be set as the View's DataContext. This can be done in many different ways, but here I have chosen to set the DataContext of the Window in XAML to a new instance of the ViewModel.

```
<Window x:Class="BMICalculator.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:BMICalculator"
        Title="BMI Calculator" Height="350" Width="525"
        FocusManager.FocusedElement="{Binding ElementName=tbxWeight}"
        FontSize="22"
        >
    <Window.DataContext>
        <local:BMIViewModel/>
    </Window.DataContext>
```

When the DataContext is set to the ViewModel it is simple to bind the TextBoxes Text property to the relevant properties:

```
        <TextBox Grid.Row="1"
                 Grid.Column="2"
                 Text="{Binding Path=Weight, StringFormat=F1}"
                 />
```

And the Button's Command property is bound to the property the returns the CalcBMICommand. Because the CalcBMICommand of the ViewModel never changes the binding Mode is set to OneTime.

```
        <Button Grid.Row="3"
                Grid.Column="1"
                Content="_Calculate BMI"
                Command="{Binding CalcBMICommand, Mode=OneTime}"
               />
```

2.3 Elaborating the BMI Calculator

TBW

Chapter

3

3 Connecting Views and ViewModels

In this chapter I will show several different ways to connect Views and ViewModels. This may sound like a simple topic, and indeed it can be, but sometimes the simple approach isn't sufficient.

The eXtreme Programming mantra: *always do the simplest thing that could possibly work* is a sound principle when you start to use the MVVM pattern. But you may soon run into trouble with the simple approach, and then you will appreciate one of the more complex techniques for connecting the View and ViewModel. One of main issues where the techniques differ is View First or ViewModel First. In the View First techniques you create a View and then the View find its ViewModel one way or another. In the ViewModel First techniques you create a ViewModel, and then the Shell/MainWindow (or some other framework component) finds the right View to connect to the ViewModel. To get started we will first look at some simple View First techniques.

3.1 View First Techniques

3.1.1 Creating a View Model in Code Behind

This is probably the simplest way to connect a ViewModel to a View. In the code behind file of a Window or UserControl you can simply initialize the DataContext property to a new instance of the relevant ViewModel as shown below.

```
public MainWindow()
{
    InitializeComponent();
    DataContext = new BMIViewModel();
}
```

Advantages:
You can send parameters with the ViewModel constructor, so you could inject the Model into the ViewModel. And you could perform some logic that could decide which ViewModel and Model that should be used in the concrete run.

Disadvantages:
Code in code behind files is difficult to unit test and you cannot give the designer some dummy data in design mode.

3.1.2 CREATING A VIEW MODEL DECLARATIVELY – V1

This is a very popular approach among the View First techniques. You simply set the DataContext of the Window or UserControl to a new instance of the ViewModel in XAML.

```
<Window x:Class="CreatingAViewModelDeclaratively.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:BMICalculator"
        Title="BMI Calculator" Height="350" Width="525"
        >
    <Window.DataContext>
        <local:BMIViewModel x:Name="viewModel"/>
    </Window.DataContext>
```

Advantages:
No code in code behind file and you can give the designer some dummy data in design mode. If you need to implement some eventhandlers in code behind, then you have access to the ViewModel by use of the name you give it in XAML.

Disadvantages:
You cannot send parameters with the ViewModel constructor.

It is difficult if you want to open a new View from a ViewModel and you want to set its ViewModel to reference some specific data in the Domain Layer (BLL) without an unwanted coupling to the View from the ViewModel.

3.1.3 CREATING A VIEW MODEL DECLARATIVELY – V2

A variation of the declarative technique is to place the ViewModel in the Resources dictionary belonging to the Wiew and then set the DataContext on a Panel by use of a StaticResource key.

```
<Window x:Class="CreatingAViewModelDeclaratively.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:BMICalculator"
        Title="BMI Calculator" Height="350" Width="525"
        FocusManager.FocusedElement="{Binding ElementName=tbxWeight}"
        FontSize="22"
        >
    <Window.Resources>
        <local:BMIViewModel x:Key="viewModel"/>
    </Window.Resources>

    <Grid DataContext="{StaticResource viewModel}">
```

Advantages:

No code in code behind file and you can give the designer some dummy data in design mode.

Disadvantages:

You cannot send parameters with the ViewModel constructor.

It is difficult if you want to open a new View from a ViewModel and you want to set its ViewModel to reference some specific data in the Domain Layer (BLL) without an unwanted coupling to the View from the ViewModel.

3.1.4 Using a ViewModel Locator

A ViewModel Locator is a class that has the responsibility of creating ViewModels. It is a simple vanilla class so you can unit test it if you so desire. And if your application requires special logic when you instantiate ViewModels the ViewModel Locator is a perfect place to put it. To achieve all the advantages of the previous techniques you connect the ViewModel to the View declaratively in XAML by use of the ViewModel Locator. We only want one instance of the ViewModel Locator in our application, and this instance should be easy to reach from all Views. One way to achieve this is by use of the Singleton pattern, but an easier approach is to put an instance of the ViewModel Locator in the Resource section of the App class (in the file App.xaml).

The code in the ViewModel Locator:

```
public class ViewModelLocator
    {
        public BMIViewModel BmiViewModel
        {
            get
            {
                return new BMIViewModel(new BMIModel());
            }
        }
    }
```

The code shown above is a very simple ViewModel Locator. They can be very smart, but often the “keep it simple” approach works fine.

To be reachable from all Views you can place the ViewModel Locator in the resources of the Application object.

In App.xaml:

```
<Application x:Class="UsingAViewModelLocator.App"
           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:UsingAViewModelLocator"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <local:ViewModelLocator x:Key="ViewModelLocator" />
    </Application.Resources>
</Application>
```

Then the View can set their DataContext declaratively:

```
<Window x:Class="UsingAViewModelLocator.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BMI Calculator" Height="350" Width="525"
        FocusManager.FocusedElement="{Binding ElementName=tbxWeight}"
        FontSize="22"
        DataContext="{Binding Source={StaticResource ViewModelLocator},
                              Path=BmiViewModel}"
        >
```

Advantages:

You can send parameters with the ViewModel constructor, so you could inject the Model into the ViewModel.

You could perform some logic that could decide which ViewModel and Model that should be used in the concrete run.

No code in code behind file.

You can give the designer some dummy data in design mode.

Disadvantages:

It is difficult if you want to open a new View from a ViewModel and you want to set its ViewModel to reference some specific data in the Domain Layer (BLL) without an unwanted coupling to the View from the ViewModel.

3.2 ViewModel First Techniques

3.2.1 A DATA TEMPLATE AS VIEW

In the resource system you can define a DataTemplate and set its DataType to a ViewModel. And then you put some kind of a Control – typical a ContentControl on the Window and use data binding to reference a property on the Windows ViewModel that gives the Control its ViewModel. The Control will then look for a DataTemplate in the Resource system to use as View.

Window.xaml:

```
    <Window.Resources>
        <DataTemplate DataType="{x:Type vm:BMIViewModel}">
            <Grid>
                <...>
            </Grid>
        </DataTemplate>
    </Window.Resources>

    <Grid>
        <ContentControl Content="{Binding Path=BmiViewModel}"
            />
    </Grid>
</Window>
```

The Window's DataContext is set to MainViewModel in App.xaml.cs (requires that you remove the StartupUri line from App.xaml):

```
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);
            Window win = new MainWindow();
            win.DataContext = new MainViewModel();
            win.Show();
        }
    }
```

And the MainViewModel has a property that returns the BMIViwModel:

```
    public class MainViewModel
    {
        public BMIViewModel BmiViewModel
        {
            get { return new BMIViewModel(new BMIModel()); }
        }
    }
```

This simple example doesn't show the full potential of this technique because it's static. The real benefit of this technique is that if the Property that the Controls binds to can return different ViewModels, then the Control will automatically pick up the corresponding View – if you have configured the DataTemplates in the resource section right.

Advantages:
When you change ViewModel the Control will automatically connect it to the corresponding View.

You can send parameters with the ViewModel constructor, so you can inject the Model into the ViewModel.

You can perform some logic that can decide which ViewModel and Model that should be used in the concrete run.

No code in code behind file – DataTemplates doesn't have codebehind files!

Disadvantages:
You cannot give the designer some dummy data in design mode.

In Visual Studio there is no support for designing DataTemplates in the Designer.

3.2.2 A USERCONTROL AS VIEW

This technique is a refinement of the previous. The only difference is that the DataTemplate consist of a UserControl. As a result a UserControl is used for the View and the DataTemplate is only used to connect a View (the UserControl) to a ViewModel.

Window.xaml:

```
<Window x:Class="UserControlAsView.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="clr-namespace:BMICalculator"
        xmlns:local="clr-namespace:UserControlAsView"
        Title="User Control as View" Height="350" Width="525">
    <Window.Resources>
        <DataTemplate DataType="{x:Type vm:BMIViewModel}">
            <local:BmiView />
        </DataTemplate>
    </Window.Resources>
    <Grid>
        <ContentControl Content="{Binding Path=BmiViewModel}"
            />
    </Grid>
</Window>
```

The UserControl that plays the role as View (the BmiView in the previous code listning):

```
<UserControl x:Class="UserControlAsView.BmiView"
             …
             >
    <Grid>
        <…>
        <Label Grid.Row="1"
               Grid.Column="1"
               HorizontalAlignment="Left"
               Target="{Binding ElementName=tbxWeight}"
               Content="_Weight:"
               />
```

The rest is the same as in the “A data template as View” example.

Advantages:
When you change ViewModel the Control will automatically connect it to the corresponding View.

You can send parameters with the ViewModel constructor, so you can inject the Model into the ViewModel.

You can perform some logic that can decide which ViewModel and Model that should be used in the concrete run.

No code in code behind file needed – but you have the opportunity if you need it.

In Visual Studio there is support for designing UserControls in the Designer.

Disadvantages:
You cannot give the designer some dummy data in design mode.

Chapter

4

4 MVVM and the N – Layer Architecture

The MVVM pattern is only concerned about the relations between the Presentation layer and the Domain layer. But that it only part of an application architecture. So how does the MVVM pattern relate to the Layered Application Architecture?

Developers tend to use the MVVM pattern differently. Partly because they see the pattern differently and partly because the context they use it in is different. There is no right and wrong way to use it. But there are some potential benefits that you may miss depending on how you apply the MVVM pattern in your application.

Here I will present the main differences in how developers apply the MVVM pattern and some issues to watch out for. If you want more variations then Jonathan Allen has a thorough description of different developer's application of the MVVM pattern in his article: So What Exactly is a View-Model? (Allen 2012).

4.1 MVVM as intended

Separation of concerns is an important feature of the MVVM pattern. The View is responsible for the graphical look of the application, the ViewModel holds the presentation logic and what else there is needed to glue the View to the Model, and the Model contains the business logic and data. And to read and write data from/to the data store (a file, a database or perhaps a remote server) we have the Data Access Layer (DAL). If you application is big you may subdivide these layers further, but I will stick to the well-known 3-layer architecture. This ideal separation of concerns gives the diagram as shown in Figure 6: The ideal diagram of the MVVM pattern in a 3-layer architecture.

One issue to watch out for is that you don't double the data. The ViewModel should only relay and perhaps convert data from the Model to the View, but the ViewModel should not store a copy of the data. If the ViewModel holds a copy of the data then you will have a harder job keeping the data in the View in sync with the data in both the ViewModel and Model.

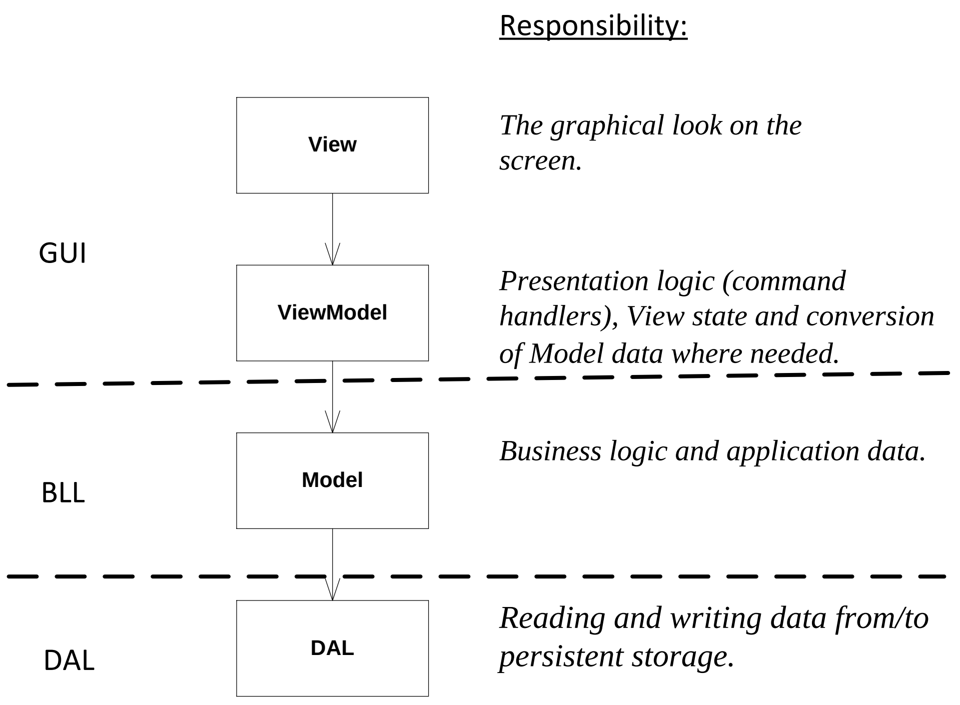
Responsibility:
View
The graphical look on the screen.
GUI
ViewModel
Presentation logic (command handlers), View state and conversion of Model data where needed.
BLL
Model
Business logic and application data.
DAL
DAL
Reading and writing data from/to persistent storage.

Figure 6: The ideal diagram of the MVVM pattern in a 3-layer architecture.

4.2 A typical implementation of MVVM

It is often preferable to validate user input close to the user. So business rules regarding data validation is often implemented in the ViewModel. And sometimes other duties like invoking methods in the DAL layer creep up in the ViewModel. In other parts of the user interface you may bypass the ViewModel and bind widgets in the View directly to objects in the Model. It is also preferable to have a Model that is without dependencies to the chosen data access technology which leads to a diagram as shown in Figure 7: A typical implementation of MVVM.

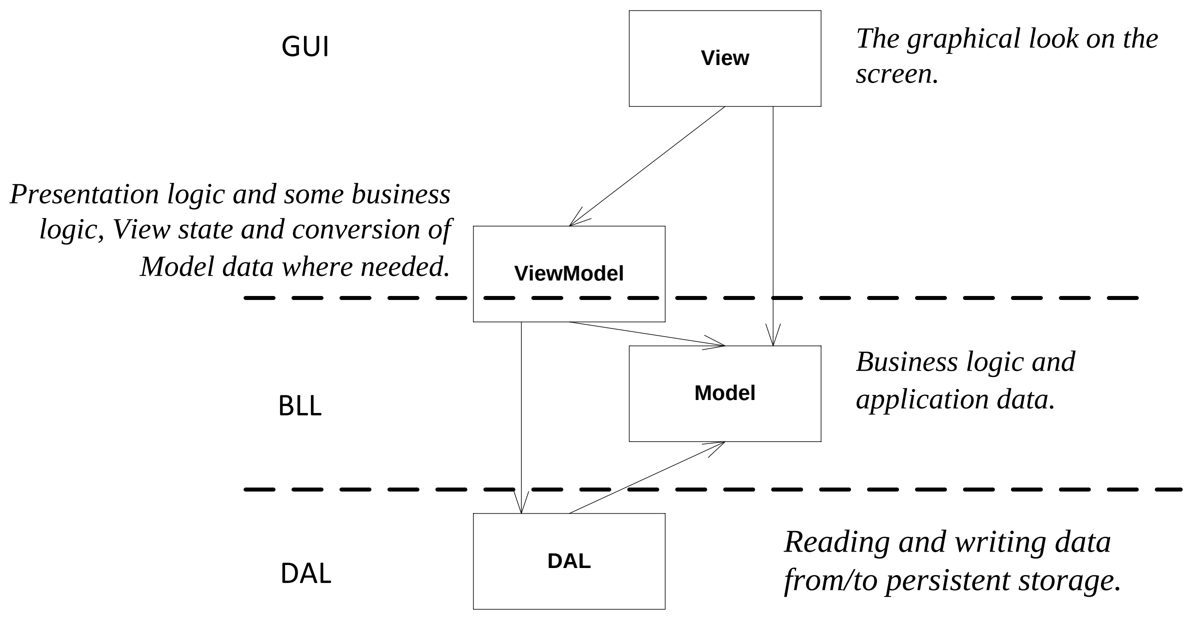
GUI
View
The graphical look on the screen.
Presentation logic and some business logic, View state and conversion of Model data where needed.
ViewModel
BLL
Model
Business logic and application data.
DAL
DAL
Reading and writing data from/to persistent storage.

Figure 7: A typical implementation of MVVM.

4.3 MVVM without a Model

Another variation is MVVM implementations without a Model. For applications that basically just edit data in a database and where there are only a few real business rules the use of a Model may be an unnecessary burden. But you must be very careful with this design as ViewModels in this design easily grows into God objects and far away from the SOLID design principles.

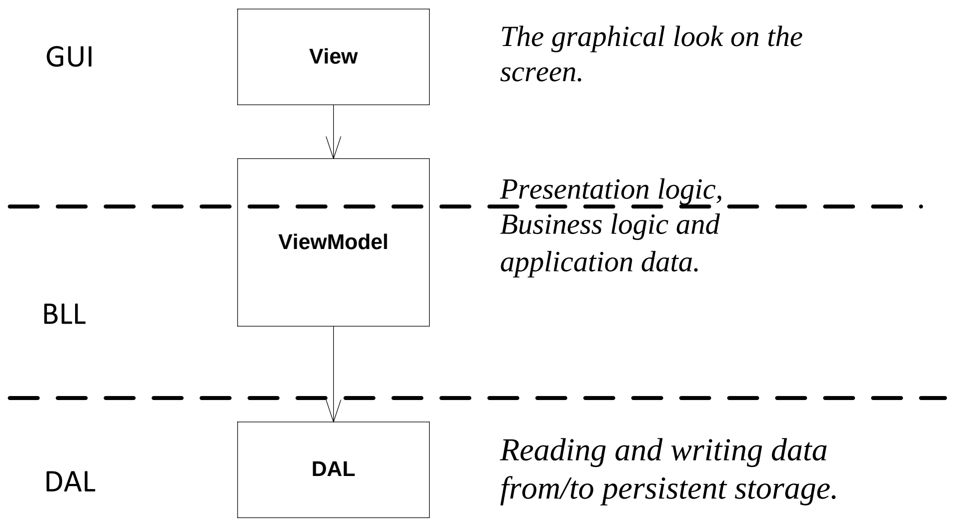
GUI
View
The graphical look on the screen.
Presentation logic, Business logic and application data.
ViewModel
BLL
DAL
DAL
Reading and writing data from/to persistent storage.

Figure 8: A MVVM implementation without Model

Chapter

5

5 MVVM Frameworks

The MVVM pattern is very popular so many developers have developed their own MVVM framework. If you do a Google search you can quickly find lots of them. Or you can look up MVVM in Wikipedia and find references to a lot of frameworks close to the bottom of the article.

Don't reinvent the wheel. It can be tempting to implement your own MVVM framework, but eventually you will have spent a lot of time on the development of your own framework. And you are likely to come up with something that is close to what is already out there.

In this chapter I will give a short presentation of a selected handful of the many MVVM frameworks. The selection is very subjective, and you may find that one of the other frameworks suit your needs better.

5.1 MVVM Foundation

Was created by Josh Smith (author of the book Advanced MVVM) and its main advantage is, that it is a very small MVVM framework making it easy to learn. Its main disadvantages are that it hasn't been updated since 2009, and it is a very minimal framework – for example you may miss a ViewModel locator implementation.

Homepage: http://mvvmfoundation.codeplex.com/ (click on the source code tab to download the framework – and build the framework yourself)

5.2 MVVM Light Toolkit

TBW.

5.3 Caliburn Micro

TBW.

5.4 Prism

TBW.

Chapter

6

6 Using MVVM light Toolkit

TBW.

Don’t reinvent the wheel.

7 Bibliography

ALLEN, J., Oct 23, 2012, 2012-last update, So What Exactly is a View-Model? [Homepage of InfoQ, C4Media Inc.], [Online]. Available: http://www.infoq.com/articles/View-Model-Definition.

BOWER, A. and MCGLASHAN, B., 2000, 2000-last update, TWISTING THE TRIAD. The evolution of the Dolphin Smalltalk MVP application framework. [Homepage of Object Arts Ltd.], [Online]. Available: http://www.object-arts.com/downloads/papers/TwistingTheTriad.PDF.

BUGNION, L., posted on Saturday, September 26, 2009 7:15 AM, 2009-last update, Using RelayCommands in Silverlight 3 and WPF [Homepage of Galasoft], [Online]. Available: http://blog.galasoft.ch/archive/2009/09/26/using-relaycommands-in-silverlight-and-wpf.aspx.

BURBECK, S., 1992, 1992-last update, Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC). Available: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P. and STAL, M., 1996. *Pattern-Oriented Software Architecture, A system of Patterns.* West Sussex, England: John Wiley & Sons Ltd.

FOWLER, M., 18 Jul 2006, 2006-last update, GUI Architectures [Homepage of martinfowler.com], [Online]. Available: http://martinfowler.com/eaaDev/uiArchs.html.

FOWLER, M., 18 Jul 06, 2006-last update, Passive View [Homepage of Martin Fowler], [Online]. Available: http://martinfowler.com/eaaDev/PassiveScreen.html.

FOWLER, M., 19 Jun 06, 2006-last update, Supervising Controller [Homepage of Martin Fowler], [Online]. Available: http://martinfowler.com/eaaDev/SupervisingPresenter.html.

FOWLER, M., 19 Jul 04, 2004-last update, Presentation Model [Homepage of Martin Fowler], [Online]. Available: http://martinfowler.com/eaaDev/PresentationModel.html.

GOSSMAN, J., 2005. *Introduction to Model/View/ViewModel pattern for building WPF apps.* USA: MSDN blog.

GREER, D., August 25, 2007, 2007-last update, Interactive Application Architecture Patterns. Available: http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/.

POTEL, M., 1996, 1996-last update, <br />MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java<br /> [Homepage of Taligent], [Online]. Available: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

REENSKAUG, T., 2003-last update, MVC*XEROX PARC 1978-79*. Available: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

SESHADRI, G., 12/29/1999, 1999-last update, Understanding JavaServer Pages Model 2 architecture [Homepage of JavaWorld.com], [Online]. Available: http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html.

SMITH, J., Feb 16, 2010 at 10:52 AM, 2010-last update, MVVM Foundation [Homepage of Codeplex], [Online]. Available: http://mvvmfoundation.codeplex.com/.

SMITH, J., February, 2009, 2009-last update, WPF Apps With The Model-View-ViewModel Design Pattern [Homepage of MSDN Magazine], [Online]. Available: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

8 Index