Introduction, evolution, data binding & programming of Advanced Model View Controller (MVC) technology - Model-View-View-Model (MVVM) with ASP.NET 4.5 and above versions

Manish Gehlot
M. Tech. Scholar, Jodhpur National University
Jodhpur, Rajasthan (India)
themanishgehlot@gmail.com

Mukul Karamchandani
JIET Group of Institutions, Jodhpur, Rajasthan (India)
mukul.karamchandani@jietjodhpur.com

Abstract

My objective of presenting this topic is making aware to programmers with this technology. It is still not accepted by programmers as expected. Still maximum programmers are working with WPF and MVC. Now it's time to work with MVVM. MVVM is advanced ASP.NET technology in which we discuss MVVM pattern and describes how to implement its fundamental characteristics. It is a part of smart web based programming. MVVM scenario is implemented using Prism Library. The MVVM pattern is a close variant of the Presentation Model pattern, optimized to leverage some of the core capabilities of WPF and Silverlight, such as data binding, data templates, commands, and behaviors. It is more reliable MVC (Model View Controller) with more advanced features that takes advantage of particular strengths of the Windows Presentation Foundation (WPF) architecture to separate the Model and the View by introducing an abstract layer between them a "Model of the View," or ViewModel. In this paper we will discuss brief introduction, features, data binding, and programming of MVVM.

Introduction

MVVM Foundation is a library of classes that are very useful when building applications based on the Model-View-ViewModel philosophy. The library is small and concentrated on providing only the most indispensable tools needed by most MVVM application developers. Model-View-ViewModel is a way of creating client applications that leverages core features of the WPF platform, allows for simple unit testing of application functionality, and helps developers and designers work together with less technical difficulties. The classes in the MVVM Foundation are time-tested tools in the toolbox of many WPF developers around the world. Now they all live in one convenient project. MvvmFoundation. The source code download also contains a set of unit tests and a demo application, which show how to use the classes. WPF lends itself toward MVVM which increase testability, flexibility and maintainability.

The Model-View-ViewModel pattern helps us to cleanly separate the business and presentation logic of our application from its user interface (UI). Maintaining a clean separation between application logic and UI helps to address numerous development and design issues and can make our application much easier to test, maintain, and evolve. It can also greatly improve code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts of the application.

Using the MVVM pattern, the UI of the application and the underlying presentation and business logic is separated into three separate classes: the view, which encapsulates the UI and UI logic; the view model, which encapsulates presentation logic and state; and the model, which encapsulates the application's business logic and data.

Prism includes samples and reference implementations that show how to implement the MVVM pattern in a Silverlight or Windows Presentation Foundation (WPF) application. The Prism Library also provides features that can help we implement the pattern in were own applications. These features embody the most common practices for implementing the MVVM pattern and are designed to support testability and to work well with Expression Blend and Visual Studio.

The pattern was first introduced by John Gossman – the architect on the Blend/WPF/Silverlight team – and derives from the Smalltalk ApplicationModel pattern, as does the PresentationModel pattern described by Martin Fowler. It was first utilized by the Expression team (John was part of that team at the time) as they developed version 1 of Expression Blend. Without the WPF/Silverlight-specific

IJERT

aspects, the Model-View-ViewModel pattern is identical to PresentationModel. In MVVM, the View and the ViewModel are typically instantiated by the container application. The View keeps a reference to the ViewModel. The ViewModel exposes commands and observable entities ("bindables") that the View binds to (or to put it in programming terms, view.DataContext = viewModel;). User interactions with the View trigger commands on the ViewModel, and updates in the ViewModel are propagated to the View through data binding. Elements of the MVVM pattern include:

Model: as in the classic MVC pattern, the model refers to either (a) a domain model which represents the real state content (an object-oriented approach), or (b) the data access layer that represents that content (a data-centric approach).

View: as in the classic MVC pattern, the view refers to all elements displayed by the GUI such as buttons, labels, and other controls.

View model: the view model is a "model of the view" meaning it is an abstraction of the view that also serves in mediating between the view and the model which is the target of the view data bindings. It could be seen as a specialized aspect of what would be a controller (in the MVC pattern) that acts as a converter that changes model information into view information and passes commands from the view into the model. The view model exposes public properties, commands, and abstractions. The view model has been likened to a conceptual state of the data as opposed to the real state of the data in the model.[10] The term "View model" is a major cause of confusion in understanding the pattern when compared to the more widely implemented MVC or MVP patterns. The role of the controller or presenter of the other patterns has been substituted with the framework binder (e.g., XAML) and view model as mediator and/or converter of the model to the binder.

Controller: some references for MVVM also include a controller layer or illustrate that the view model is a specialized functional set in parallel with a controller, while others do not. This difference is an ongoing area of discussion regarding the standardization of the MVVM pattern.

Binder: the use of a declarative data binding and command bind technology is an implicit part of the pattern. Within the Microsoft stack this is the XAML technology.[11] Essentially this architectural component frees the developer from being obliged to write boiler plate logic to synchronize the view model and view. It is the organization of code to make best use of this capability which distinguishes the pattern from both MVC and MVP. When implemented outside of the Microsoft stack the presence of a declarative data binding technology is a key enabler of the pattern.

MVVM from beginning

Since people began to create user interfaces for software were popular design patterns to help make it easier. For example, the Model-View-Presenter pattern (MVP), has gained popularity in different programming platforms IU. MVP is a variant of the Model-View-Controller pattern (MVP), which has been around for decades. In case we have never used before MVP model, here is a simplified explanation. What we see on screen is the view, which displays the data is a model and presenter joins the two together. View relies on a presenter to populate the data model, react to user input, ensure input validation (perhaps by delegating to the model), and other such tasks.

Back in 2004, Martin Fowler published an article about a pattern called Presentation Model (PM). AM pattern is similar to MVP, in that it separates a view of behavior and its status. The interesting pattern of PM is that a view is created by abstraction, called PM. One view, then, becomes one of PM's rendering. In Fowler's explanation, he shows that a PM's frequently updates its view, so that the two are in synchronization with each other. This synchronous logic code exists as Presentation Model classes. In 2005, John Gossman, one of the architects of WPF and Microsoft Silverlight on his blog reveals pattern Model-View-ViewModel (MVVM). MVVM presentation is the same model (PM) presented by Fowler, is that both models have that feature an abstraction of a View, which contains state and behavior of View. AM Fowler brought a meaning to create a platform independent graphical interface View abstractizand a while Gossman MVVM introduced as a standardized way to influent basic features of WPF thus simplifying the creation of interfaces. In this sense, I think MVVM pattern as a specialization of more general AM

IJERT

pattern in order for WPF and Silverlight platforms. In the articles of Glenn Block's "Prism: Patterns for Building Composite Applications with WPF", he explains the guidelines for Microsoft Composite Application for WPF. ViewModel only term used. But word of Presentation Mode is used to describe an abstract view. In this project, I mean I MVVM pattern and abstraction of the view that a ViewModel. I find this terminology more common in books about WPF and the Silverlight community. In contrast to the Presenter in MVP, a ViewModel not need a reference to a view. A view of ViewModel assigned properties, which, in turn, expose data models and objects contained in other state-specific view. ViewModel award of view and are easy to build as one object is assigned as a DataContext ViewModel for that view. If properties in ViewModel values change, those new values will propagate the view is automatically our data via the award. When the user clicks on a button in the View, a command is executed ViewModel action necessary to achieve that. ViewModel, never View, realizaeaza all changes made to the data model.

The Evolution of Model-View-View-Model

Back in 2004, Martin Fowler published an article about a pattern named Presentation Model (PM). The PM pattern is similar to MVP in that it separates a view from its behavior and state. The interesting part of the PM pattern is that an abstraction of a view is created, called the Presentation Model. A view, then, becomes merely a rendering of a Presentation Model. In Fowler's explanation, he shows that the Presentation Model frequently updates its View, so that the two stay in sync with each other. That synchronization logic exists as code in the Presentation Model classes. In 2005, John Gossman, currently one of the WPF and Silverlight Architects at Microsoft, unveiled theModel-View-ViewModel (MVVM) pattern on his blog. MVVM is identical to Fowler's Presentation Model, in that both patterns feature an abstraction of a View, which contains a View's state and behavior. Fowler introduced Presentation Model as a means of creating a UI platform-independent abstraction of a View, whereas Gossman introduced MVVM as a standardized way to leverage core features of WPF to simplify the creation of user interfaces. In that sense, I consider MVVM to be a specialization of the more general PM pattern, tailor-made for the WPF and Silverlight platforms.

In Glenn Block's excellent article "Prism: Patterns for Building Composite Applications with WPF" in the September 2008 issue, he explains the Microsoft Composite Application Guidance for WPF. The term ViewModel is never used. Instead, the term Presentation Model is used to describe the abstraction of a view. Throughout this article, however, I'll refer to the pattern as MVVM and the abstraction of a view as a ViewModel. I find this terminology is much more prevelant in the WPF and Silverlight communities.

Unlike the Presenter in MVP, a ViewModel does not need a reference to a view. The view binds to properties on a ViewModel, which, in turn, exposes data contained in model objects and other state specific to the view. The bindings between view and ViewModel are simple to construct because a ViewModel object is set as the DataContext of a view. If property values in the ViewModel change, those new values automatically propagate to the view via data binding. When the user clicks a button in the View, a command on the ViewModel executes to perform the requested action. The ViewModel, never the View, performs all modifications made to the model data.

The view classes have no idea that the model classes exist, while the ViewModel and model are unaware of the view. In fact, the model is completely oblivious to the fact that the ViewModel and view exist. This is a very loosely coupled design, which pays dividends in many ways, as we will soon see.

Why MVVM?

Once a developer becomes comfortable with WPF and MVVM, it can be difficult to differentiate the two. MVVM is the lingua franca of WPF developers because it is well suited to the WPF platform, and WPF was designed to make it easy to build applications using the MVVM pattern (amongst others). In fact, Microsoft was using MVVM internally to develop WPF applications, such as Microsoft Expression Blend, while the core WPF platform was under construction. Many aspects of WPF, such as the look-less control model and data templates, utilize the strong separation of display from state and behavior promoted by MVVM.

The single most important aspect of WPF that makes MVVM a great pattern to use is the data binding infrastructure. By binding properties of a view to a ViewModel, we get loose coupling between the two and entirely remove the need for writing code in a ViewModel that directly updates a view. The data binding system also supports input validation, which provides a standardized way of transmitting validation errors to a view.

Two other features of WPF that make this pattern so usable are data templates and the resource system. Data templates apply Views to ViewModel objects shown in the user interface. We can declare templates in XAML and let the resource system automatically locate and apply those templates for we at run time. We can learn more about binding and data templates in my July 2008 article, "Data and WPF: Customize Data Display with Data Binding and WPF."

If it were not for the support for commands in WPF, the MVVM pattern would be much less powerful. In this article, I will show we how a ViewModel can expose commands to a View, thus allowing the view to consume its functionality. If we aren't familiar with commanding, I recommend that we read Brian Noyes's comprehensive article, "Advanced WPF: Understanding Routed Events and Commands in WPF," from the September 2008 issue.

In addition to the WPF (and Silverlight 2) features that make MVVM a natural way to structure an application, the pattern is also popular because ViewModel classes are easy to unit test. When an application's interaction logic lives in a set of ViewModel classes, we can easily write code that tests it. In a sense, Views and unit tests are just two different types of ViewModel consumers. Having a suite of tests for an application's ViewModels provides free and fast regression testing, which helps reduce the cost of maintaining an application over time.

In addition to promoting the creation of automated regression tests, the testability of ViewModel classes can assist in properly designing user interfaces that are easy to skin. When we are designing an application, we can often decide whether something should be in the view or the ViewModel by imagining that we want to write a unit test to consume the ViewModel. If we can write unit tests for the ViewModel without creating any UI objects, we can also completely skin the ViewModel because it has no dependencies on specific visual elements.

Lastly, for developers who work with visual designers, using MVVM makes it much easier to create a smooth designer/developer workflow. Since a view is just an arbitrary consumer of a ViewModel, it is easy to just rip one view out and drop in a new view to render a ViewModel. This simple step allows for rapid prototyping and evaluation of user interfaces made by the designers.

The development team can focus on creating robust ViewModel classes, and the design team can focus on making user-friendly Views. Connecting the output of both teams can involve little more than ensuring that the correct bindings exist in a view's XAML file.

Data Binding with MVVM

Data binding plays a very important role in the MVVM pattern. WPF and Silverlight both provide powerful data binding capabilities. Our view model and (ideally) our model classes should be designed to support data binding so that they can take advantage of these capabilities. Typically, this means that they must implement the correct interfaces.

Silverlight and WPF data binding supports multiple data binding modes. With one-way data binding, UI controls can be bound to a view model so that they reflect the value of the underlying data when the display is rendered. Two-way data binding will also automatically update the underlying data when the user modifies it in the UI.

To ensure that the UI is kept up to date when the data changes in the view model, it should implement the appropriate change notification interface. If it defines properties that can be data bound, it should implement the **INotifyPropertyChanged** interface. If the view model represents a collection, it should implement the **INotifyCollectionChanged**interface or derive from the **ObservableCollection<T>** class that provides an implementation of this interface. Both of these interfaces define an event that is raised whenever the underlying data is changed. Any data bound controls will be automatically updated when these events are raised.

In many cases, a view model will define properties that return objects (and which, in turn, may define properties that return additional objects). WPF and Silverlight data binding supports binding to nested properties via the **Path** property.

IJERT

Therefore, it is very common for a view's view model to return references to other view model or model classes. All view model and model classes accessible to the view should implement the **INotifyPropertyChanged** or **INotifyCollectionChanged** interfaces, as appropriate.The following sections describe how to implement the required interfaces in order to support data binding within the MVVM pattern.

The MVVM pattern helps we to cleanly separate our UI from our presentation and business logic and data, so implementing the right code in the right class is an important first step in using the MVVM pattern effectively. Managing the interactions between the view and view model classes through data binding and commands are also important aspects to consider. The next step is to consider how the view, view model, and model classes are instantiated and associated with each other at run time.

Typically, there is a one-to-one relationship between a view and its view model. The view and view model are loosely coupled via the view's data context property; this allows visual elements and behaviors in the view to be data bound to properties, commands, and methods on the view model. We will need to decide how to manage the instantiation of the view and view model classes and their association via the **DataContext** property at run time.

Care must also be taken when constructing and connecting the view and view model to ensure that loose coupling is maintained. As noted in the previous section, the view model should ideally not depend on any specific implementation of a view. Similarly, the view should ideally not depend on any specific implementation of a view model.

There are multiple ways the view and the view model can be constructed and associated at run time. The most appropriate approach for our application will largely depend on whether we create the view or the view model first, and whether we do this programmatically or declaratively. The following sections describe common ways in which the view and view model classes can be created and associated with each other at run time.

Data Validation and Error Reporting

Our view model or model will often be required to perform data validation and to signal any data validation errors to the view so that the user can act to correct them.

Silverlight and WPF provide support for managing data validation errors that occur when changing individual properties that are bound to controls in the view. For single properties that are data-bound to a control, the view model or model can signal a data validation error within the property setter by rejecting an incoming bad value and throwing an exception. If the ValidatesOnExceptions property on the data binding is true, the data binding engine in WPF and Silverlight will handle the exception and display a visual cue to the user that there is a data validation error.

However, throwing exceptions with properties in this way should be avoided where possible. An alternative approach is to implement the IDataErrorInfo orINotifyDataErrorInfo interfaces on our view model or model classes. These interfaces allow our view model or model to perform data validation for one or more property values and to return an error message to the view so that the user can be notified of the error.

Creating View Model using XAML and by programmatically

Perhaps the simplest approach is for the view to declaratively instantiate its corresponding view model in XAML. When the view is constructed, the corresponding view model object will also be constructed. We can also specify in XAML that the view model be set as the view's data context.

The XAML-based approach is demonstrated in the QuestionnaireView.xaml file in the Basic MVVM QuickStart. In that example, the QuestionnaireViewModel instance is defined in the QuestionnaireView's XAML, as shown here.

XAML

```
<UserControl.DataContext>
   <my:QuestionnaireViewModel/>
</UserControl.DataContext>
```

When the **QuestionnaireView** is created, an instance of the **QuestionnaireViewModel** is automatically constructed and set as the view's data context. This approach requires wer view model to have a default (parameter-less) constructor.

The declarative construction and assignment of the view model by the view has the advantage that it is simple and works well in design-time tools such as Microsoft Expression Blend or Microsoft Visual Studio. The disadvantage of this approach is that the view has knowledge of the corresponding view model type.

IJERT

An approach is for the view to instantiate its corresponding view model instance programmatically in its constructor. It can then set it as its data context, as shown in the following code example.

C#

```
public QuestionnaireView()
{
   InitializeComponent();
   this.DataContext = new QuestionnaireViewModel();
}
```

The programmatic construction and assignment of the view model within the view's code-behind has the advantage that it is simple and works well in design-time tools like Expression Blend or Visual Studio. The disadvantage of this approach is that the view needs to have knowledge of the corresponding view model type and that it requires code in the view's code-behind. Using a dependency injection container, such as Unity or MEF, can help to maintain loose coupling between the view and view model.

Future Scope & Development

For every new technology there are some common future scopes like removing bugs, making technology more reliable and user friendly etc. Still MVVM technology is not adapted by programmers as expected by developers of Microsoft. So the most important objective will be making it more familiar to programmers. There may be one more reason that still many programmers are using the old technology because they have perfection with that one. For example still people are using Windows 7. Still very few people have Windows 8 and even Microsoft is planning for Windows 9. So when Miscrosoft will launch Windows 9, will then people will accept Windows 8? So getting reasons why people are not friendly with advanced technology will be future scope of MVVM. When programmers use the technology then we are able to know what we need more in this technology.

References

[1] C# 3.0 Design Patterns, Judith Bishop, O'Reilly, 2007

[2] Introduction to Design Patterns in C# and WPF, James W. Cooper, 2002, IBM T J Watson Research Center

[3] WPF patterns: MVC, MVP or MVVM, http://www.orbifold.net/default/?p=550

[4] MVVM, a WPF UI Design Pattern, http://channel9.msdn.com/shows/Continuum/MVVM/

[5] Model View ViewModel, http://en.wikipedia.org/wiki/Model_View_ViewModel

[6] MVVM Foundation, http://en.wikipedia.org/wiki/Model_View_ViewModel

[7] http://msdn.microsoft.com/en-us/library/ms752347.aspx

[8] http://msdn.microsoft.com/en-us/library/ms742521.aspx.

[9] Advanced MVVM by Mr. Josh Smith, Senior UX Developer at IdentityMine

[10] MVVM tutorial with WPF, wpftutorial.net/MVVM.html

[11] MVVM Light Toolkit, mvvmlight.codeplex.com

[12] MVVM Foundation, mvvmfoundation.codeplex.com

[13] http://www.codeproject.com/Articles/186705/A-Totally-Simple-Introduction-to-the-MVVM-Concept

[14] http://msdn.microsoft.com/en-us/magazine/dd19663.aspx

[15] http://russelleast.wordpress.com/2008/08/09/overview-of-the-modelview-viewmodel-mvvm-pattern-and-data-binding/

IJERT