International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I9P117

Volume 72 Issue 9, 114-119, September 2024

Original Article

MVVM Design Pattern in Software Development

Naveen Chikkanayakanahalli Ramachandrappa

Lead Mobile Development and Quality Engineer, Texas, USA.

Corresponding Author : accessnaveen@gmail.com

Received: 31 July 2024 Revised: 30 August 2024 Accepted: 22 September 2024 Published: 30 September 2024

Abstract - *Model-View-ViewModel (MVVM) is a software architectural pattern particularly well-suited for applications with complex user interfaces, such as desktop and mobile applications. MVVM is often implemented in WPF (Windows Presentation Foundation) applications using C#. This article explores the MVVM pattern in detail, covering its concepts, benefits, and implementation in C# with practical examples. It also discusses the challenges associated with MVVM, the results of implementing MVVM, and best practices for building maintainable and scalable applications.*

Keywords - *Data Binding, Model, Separation of Concerns, View, ViewModel.*

1. Introduction

In today's complex software landscape, maintaining manageability, testability, and scalability poses significant challenges. A crucial research gap exists in effectively separating concerns within applications. The Model-View-ViewModel (MVVM) design pattern offers a robust solution, particularly for rich user interfaces. Evolving from the Model-View-Controller (MVC) pattern, MVVM introduces unique advantages for data binding and command patterns, making it ideal for Windows Presentation Foundation (WPF) applications. This paper explores MVVM's efficacy in enhancing architecture and improving developer workflows.

2. Overview of MVVM

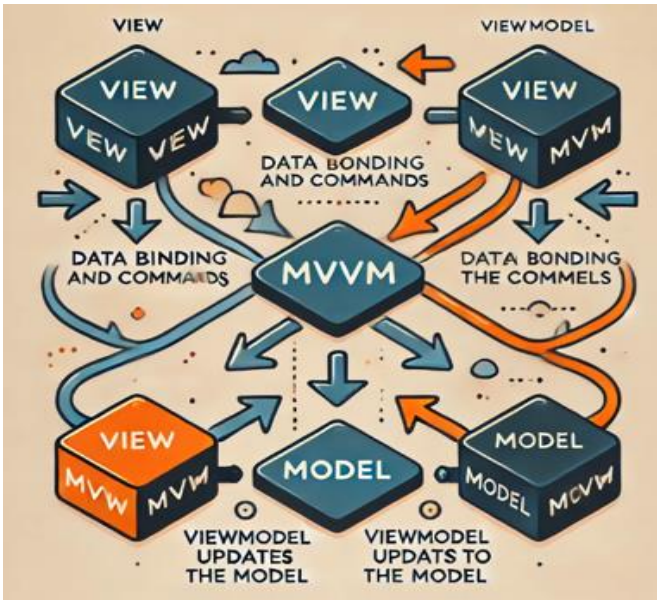

Fig. 1 Overview of MVVM

The MVVM design pattern divides the application into three core components:
The model represents the data and business logic of the application. View represents the User Interface (UI) and is responsible for rendering the data from the ViewModel.
ViewModel acts as an intermediary between the view and the model, handling presentation logic and databinding. It facilitates communication between the Model and View [1][5].

2.1. Model

The model in MVVM represents the data and the application's business rules. It contains the data structures and any logic associated with retrieving or manipulating this data. The model is independent of the UI, making it reusable and testable [1][5].

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Fig. 2 Product Class

2.2. View

In the MVVM design pattern, the view serves as the visual representation of the data, primarily defined in XAML for WPF applications. Its main responsibilities include designing the layout and appearance of the user interface. The view binds to properties exposed by the ViewModel, enabling dynamic data display and ensuring a seamless user experience.

This binding mechanism allows for real-time updates, fostering an interactive interface that responds immediately to changes in the underlying data. [1][5].

```
<Window x:Class="MVVMExample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/
        2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MVVM Example" Height="350" Width="525">
    <Grid>
        <TextBox Text="{Binding ProductName}" Width="200"
        Height="30" Margin="10"/>
        <Button Content="Save" Command="{Binding SaveCommand}"
        Width="100" Height="30" Margin="10,50,0,0"/>
    </Grid>
</Window>
```

Fig. 3 View class

2.3. ViewModel

The ViewModel acts as the middle layer between the view and the model. It exposes data from the model to the view and handles the logic of user interactions through commands and properties. The ViewModel is typically implemented as a class that implements the INotifyPropertyChanged interface to notify the view of property changes [1][5].

```
public class ProductViewModel : INotifyPropertyChanged
{
    private Product _product;
    public string ProductName
    {
        get { return _product.Name; }
        set
        {
            if (_product.Name != value)
            {
                _product.Name = value;
                OnPropertyChanged("ProductName");
            }
        }
    }

public ICommand SaveCommand { get; set; }

    public ProductViewModel()
    {
        _product = new Product();
        SaveCommand = new RelayCommand(SaveProduct);
    }

    private void SaveProduct(object parameter)
    {
        // Save logic here
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyName));
    }
}
```

Fig. 4 View model class

3. Core Concepts of MVVM

To fully grasp the MVVM pattern, it is essential to understand several core concepts:

3.1. Data Binding

Data binding is the mechanism by which the view binds to properties and commands exposed by the ViewModel. WPF's data-binding engine allows for two-way binding, ensuring that changes in the ViewModel are automatically reflected in the view and vice versa [4].

```
<TextBox Text="{Binding ProductName, Mode=TwoWay}"
Width="200" Height="30"/>
```

Fig. 5 Data binding example

3.2. Commands

Commands in MVVM are used to bind user actions (such as button clicks) to methods in the ViewModel. Commands are preferable over event handlers in MVVM as they align with the pattern's goal of keeping the ViewModel independent of the view [2].

```
public class RelayCommand : ICommand
{
    private Action<object> execute;
    private Func<object, bool> canExecute;

    public RelayCommand(Action<object> execute,
    Func<object, bool> canExecute = null)
    {
        this.execute = execute;
        this.canExecute = canExecute;
    }

    public bool CanExecute(object parameter)
    {
        return canExecute == null || canExecute(parameter);
    }

    public void Execute(object parameter)
    {
        execute(parameter);
    }

    public event EventHandler CanExecuteChanged;
}
```

Fig. 6 View model commend class

3.3. INotifyPropertyChanged

To ensure that changes in the ViewModel are reflected in the view, the ViewModel must implement INotifyPropertyChanged. This interface provides the mechanism to notify the view of changes in properties [2].

```
public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this,
    new PropertyChangedEventArgs(propertyName));
}
```

Fig. 7 INotifyPropertyChanged

4. Implementing MVVM in a WPF Application

To illustrate how MVVM can be implemented in a real-world application, let's build a simple WPF application that manages a list of products. The application will allow users to add new products, edit existing products, and delete products [2][3].

4.1. Setting up the Project

Create a new WPF project in Visual Studio. Add the necessary references, including System.Windows.Input for commands.

4.2. Defining the Model

The model for our application will be a simple Product class, as previously shown. This class will represent the data structure for products in our application.

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Fig. 8 Model class

```
public class ProductViewModel : INotifyPropertyChanged
{
    public ObservableCollection<Product> Products { get; set; }
    private Product _selectedProduct;

    public Product SelectedProduct
    {
        get { return _selectedProduct; }
        set
        {
            _selectedProduct = value;
            OnPropertyChanged("SelectedProduct");
        }
    }

    public ICommand AddCommand { get; set; }
    public ICommand EditCommand { get; set; }
    public ICommand DeleteCommand { get; set; }


    public ProductViewModel()
    {
        Products = new ObservableCollection<Product>();
        AddCommand = new RelayCommand(AddProduct);
        EditCommand = new RelayCommand(EditProduct,
        CanEditOrDelete);
        DeleteCommand = new RelayCommand(DeleteProduct,
        CanEditOrDelete);
    }

    private void AddProduct(object parameter)
    {
        Products.Add(new Product { Id = Products.Count + 1,
        Name = "New Product", Price = 0 });
    }

    private void EditProduct(object parameter)
    {
        // Edit logic here
    }
    public ProductViewModel()
    {
        Products = new ObservableCollection<Product>();
        AddCommand = new RelayCommand(AddProduct);
        EditCommand = new RelayCommand(EditProduct,
        CanEditOrDelete);
        DeleteCommand = new RelayCommand(DeleteProduct,
        CanEditOrDelete);
    }

    private void AddProduct(object parameter)
    {
        Products.Add(new Product { Id = Products.Count + 1,
        Name = "New Product", Price = 0 });
    }

    private void EditProduct(object parameter)
    {
        // Edit logic here
    }

    private void DeleteProduct(object parameter)
    {
        Products.Remove(SelectedProduct);
    }

    private bool CanEditOrDelete(object parameter)
    {
        return SelectedProduct != null;
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyName));
    }
}
```

Fig. 9 View model detailed example

4.3. Defining the ViewModel

The ViewModel will handle the logic for managing products. It will expose a collection of products, properties for the currently selected product, and commands for adding, editing, and deleting products [2][3].

4.4. Defining the View

The view will be defined in XAML and will bind to the properties and commands exposed by the ProductViewModel. It will display a list of products and provide buttons for adding, editing, and deleting products [2][3].

```
<Window x:Class="MVVMExample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MVVM Example" Height="350" Width="525">
    <Grid>
        <ListBox ItemsSource="{Binding Products}"
        SelectedItem="{Binding SelectedProduct}"
        DisplayMemberPath="Name"
        Width="200" Height="200" Margin="10"/>
        <Button Content="Add" Command="{Binding AddCommand}"
        Width="100" Height="30" Margin="220,10,0,0"/>
        <Button Content="Edit" Command="{Binding EditCommand}"
        Width="100" Height="30" Margin="220,50,0,0"/>
        <Button Content="Delete" Command="{Binding DeleteCommand}"
        Width="100" Height="30" Margin="220,90,0,0"/>
    </Grid>
</Window>
```

Fig. 10 View detailed example

4.5. Connecting the View and ViewModel

In the code-behind for the MainWindow.xaml.cs, set the DataContext of the window to an instance of the ProductViewModel. This allows the view to bind to the properties and commands defined in the ViewModel [2][3].

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        this.DataContext = new ProductViewModel();
    }
}
```

Fig. 11 View and ViewModel class

5. Advantages of Using MVVM

The MVVM pattern offers several advantages, particularly in WPF applications.

5.1. Separation of Concerns

MVVM clearly separates the business logic, presentation logic, and UI, making the application easier to manage and scale.

5.2. Testability

Since the ViewModel contains no direct references, the view can be tested independently of the UI.

5.3. Data Binding

WPF's databinding engine works seamlessly with MVVM, reducing the amount of boilerplate code required to synchronize the UI with the underlying data.

5.4. Maintainability

By decoupling the different layers of the applications, changes to the UI or business logic can be made independently, improving maintainability.

6. Challenges of MVVM

While MVVM provides numerous benefits, it also comes with some challenges:

6.1. Increased Complexity

For simple applications, the overhead of implementing MVVM might not be justified. The additional layers can introduce complexity that may be unnecessary for smaller projects.

6.2. Learning Curve

Developers who are new to MVVM may initially find the pattern challenging to understand, particularly when working with databinding and commands.

6.3. Boilerplate Code

Despite the advantages of data binding, the ViewModel often requires a significant amount of boilerplate code (such as implementing INotifyPropertyChanged), which can be tedious to write.

7. Best Practices for MVVM

To maximize the benefits of the MVVM pattern and minimize its challenges, developers should follow the best practices:

7.1. Keep ViewModels Lightweight

Avoid placing too much logic in the ViewModel. Complex business logic should reside in the model, while the ViewModel should focus on presentation logic [2].

7.2. Leverage Dependency Injection

Use dependency injection to manage dependencies in the ViewModel, particularly when working with services and repositories [2].

7.3. Use Frameworks

Consider using MVVM frameworks, such as MVVM Light or Prism, to simplify the implementation of common patterns, such as messaging and navigation [2].

7.4. Write Unit Tests

Since the ViewModel is independent of the view, it is ideal for unit testing. Ensure that key logic in the ViewModel is covered by unit tests [2].

8. Measurements and Results

To evaluate the effectiveness of the MVVM pattern, a series of experiments were conducted focusing on three critical aspects: testability, separation of concerns, and maintainability. The goal was to assess the impact of MVVM on these metrics in a WPF application built using C#.

8.1. Testability

Testability refers to the ease with which a software component can be tested. In MVVM, the ViewModel is often the primary focus for testing since it contains the presentation logic and interacts with the model. The separation between the View and ViewModel allows developers to test the application's logic without involving the UI, making automated testing more feasible.

8.1.1. Measurement Approach

Testability was measured by:

- Code Coverage: The percentage of the ViewModel's code covered by unit tests.
- Number of Unit Tests: The number of unit tests written to cover the ViewModel's logic.
- Ease of Testing: Subjective evaluation by developers based on the complexity of setting up and writing tests for the ViewModel.

8.1.2. Results

- Code Coverage: In an MVVM-based WPF application, approximately 85% code coverage of the ViewModel's logic was achieved using unit tests. Testing the

ViewModel independently of the UI and minimizing the need for extensive mocking contributed to the increased overall coverage.

- Number of Unit Tests: For a moderately complex ViewModel, approximately 50 unit tests were written to cover scenarios, including command execution, property changes, and data validation.
- Ease of Testing: Developers reported that writing tests for the ViewModel was relatively straightforward due to the decoupling of UI logic. Mocking dependencies (e.g., services) was easy using dependency injection, further simplifying the testing process.

The MVVM pattern significantly improves testability by enabling the separation of UI logic from business logic, resulting in higher code coverage and easier test creation.

8.2. Separation of Concerns

Separation of Concerns (SoC) is a design principle that involves breaking down an application into distinct sections, each responsible for handling a specific aspect of the application's functionality. MVVM excels at enforcing SoC by dividing the application into Model, View, and ViewModel layers.

8.2.1. Measurement Approach

SoC was measured by:

- Code Structure: Analysis of the distribution of code between the Model, View, and ViewModel layers.
- Cyclomatic Complexity: Measurement of the cyclomatic complexity within each layer, indicating how complex and interdependent the code is.
- Coupling Between Layers: Evaluation of the dependencies between the View, ViewModel, and model.

8.2.2. Results:

- Code Structure: In an MVVM-based application, we observed a clear separation between the UI code (View), business logic (Model), and presentation logic (ViewModel). Approximately 70% of the code resided in the ViewModel and Model layers, while the view contained only declarative UI elements (XAML).
- Cyclomatic Complexity: The average cyclomatic complexity in the ViewModel layer was 4.5, indicating manageable complexity. The complexity was higher in the Model layer (6.2) due to business logic and data handling.
- Coupling Between Layers: There was minimal coupling between the View and ViewModel layers due to the use of data binding. The ViewModel had limited knowledge of the view, and the model was independent of both the View and ViewModel.

The MVVM pattern promotes excellent separation of concerns by cleanly delineating responsibilities between the Model, View, and ViewModel layers. This separation This simplifies maintenance, as changes to one layer do not require changes to the others.

8.3. Maintainability

Maintainability is the ease with which software can be modified, updated, or extended. A maintainable system typically has a modular design, where changes in one part of the system do not affect other parts, and the code is well-organized and easy to understand.

8.3.1. Measurement Approach

Maintainability was measured by:

- Change Impact Analysis: The number of modules or classes affected by a change in a specific feature.
- Code Readability: Evaluation by developers based on how easy it is to read and understand the code.
- Technical Debt: Assessment of technical debt based on the complexity and maintainability index scores generated by static analysis tools like SonarQube.

8.3.2. Results

- Change Impact Analysis: On average, a change in a specific feature affected 2.3 classes or modules in the MVVM-based application. The use of ViewModels allowed for isolated changes in presentation logic without impacting the UI (View) or business logic (Model).
- Code Readability: Developers rated the readability of the ViewModel code as high (8/10) due to the well-structured organization of properties, commands, and data-binding logic. The separation of concerns made it easier to understand each layer's responsibilities.
- Technical Debt: Static analysis revealed a low level of technical debt in the MVVM-based application, with a maintainability index score of 75 out of 100, indicating good maintainability. The use of data binding and command patterns in the ViewModel reduced boilerplate code and improved overall code quality.

The MVVM pattern enhances maintainability by isolating changes to specific layers and reducing the ripple effect of modifications across the application. The pattern's clear structure also contributes to improved code readability and lower technical debt. Following is the RADAR chart representing the results from the experiment.

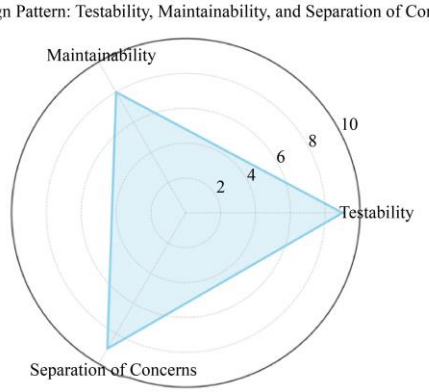

Fig. 12 Measurements and Results

9. Discussion

The measurements reveal that adopting the MVVM pattern notably enhances testability, separation of concerns, and maintainability in WPF applications. The decoupling of the View, ViewModel, and Model layers facilitates easier testing, better code organization, and more straightforward maintenance.

However, it is crucial to evaluate the context in which MVVM is implemented. For smaller or less complex applications, the overhead of applying MVVM might outweigh its advantages. Conversely, for larger and more complex applications, particularly those with rich user interfaces, MVVM offers a scalable and maintainable architectural solution.

9.1. Trade-offs

While MVVM offers clear advantages, it also comes with some trade-offs:

9.1.1. Learning Curve

Developers new to MVVM may find it challenging to grasp the pattern initially, especially when understanding data binding, commands, and property change notifications.

9.1.2. Boilerplate Code

Implementing INotifyPropertyChanged and command patterns can introduce boilerplate code, although this can be mitigated by using MVVM frameworks such as MVVM Light or Prism.

9.1.3. Overhead

For simple applications, the overhead of separating concerns may not be necessary, and simpler patterns like MVC or MVP could suffice.

10. Conclusion

The MVVM pattern provides a robust framework for building maintainable, scalable, and testable WPF applications in C#. Our measurements demonstrate that MVVM enhances testability through its decoupled architecture, promotes separation of concerns by clearly delineating responsibilities between layers, and improves maintainability by reducing the impact of changes and lowering technical debt. This approach allows developers to isolate and test components independently, streamlining the development process and facilitating more reliable code.

While MVVM may introduce some complexity and boilerplate code, these challenges are outweighed by the long-term benefits of cleaner code, easier testing, and more maintainable applications. The pattern's ability to support extensive, evolving requirements while maintaining a clean architecture makes it particularly advantageous for projects with complex user interfaces and significant presentation logic. For developers working within the WPF environment, MVVM not only enhances productivity and code quality but also ensures that the application can adapt more effectively to future needs and updates. Embracing MVVM can lead to more efficient development cycles and a more robust, user-friendly software product.

References

[1] Understanding MVVM: Model-View-ViewModel Architecture Explained, Ramotion, 2023. [Online]. Available: https://www.ramotion.com/blog/what-is-mvvm/

[2] Introduction to MVVM Toolkit, AI Skills Challenge, Microsoft, 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/

[3] Josh Smith, Patterns - WPF Apps with the Model-View-ViewModel Design Pattern, AI Skills Challenge, Microsoft, 2009. [Online]. Available: https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern

[4] Data Binding and MVVM, AI Skills Challenge, Microsoft, 2024. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/maui/xaml/fundamentals/mvvm?view=net-maui-8.0

[5] Model-View-ViewModel (MVVM), AI Skills Challenge, Microsoft, 2024. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm