C#2.0 .NET3.0 C# Windows .NET XAML WPF MVC MVVM

Model View Controller, Model View Presenter, and Model View ViewModel Design Patterns

alexy.shelest

Rate me: 4.79/5 (63 votes)

3 Oct 2009 CPOL 12 min read 358.4K 12 192 23

This article will compare and contrast the MVC, MVP, and MVVM, and suggest which pattern to use based on your technology of choice and the problem that you are trying to solve.

Introduction

The recent growth in UI-centric technologies has refueled interest in presentation layer design patterns. One of the most quoted patterns, Model-View-Controller (MVC), is thought to be designed by Trygve Reenskaug, a Norwegian computer engineer, while working on Smalltalk-80 in 1979 [1]. It was subsequently described in depth in the highly influential “Design Patterns: Elements of Reusable Object-Oriented Software” [2], a.k.a. the “Gang of Four” book, in 1994. Two years later, Mike Potel from Taligent (IBM) published his paper, “Model-View-Presenter (MVP) - The Taligent Programming Model for C++ and Java” [3], where he aimed to address the shortfalls of the MVC. Both MVP and MVC have since been widely adopted by Microsoft in their Composite Application Blocks (CAB) and the ASP.NET MVC frameworks.

These patterns and their kinship were back on the radar in 2004 when Martin Fowler analysed them in his papers. He suggested splitting them into other smaller patterns (Supervising Controller and Passive View), and unveiled his solution to this problem in his paper “Presentation Model (PM)” [6]. With the release of the Windows Presentation Foundation (WPF), the new term, Model-View-

ViewModel, has entered the scene. First mentioned by the WPF Architect, John Gossman, on his blog in 2005 [7], it was later described by Josh Smith in the MSDN article “WPF Apps with the Model-View-ViewModel Design Pattern” [8]. MVVM was built around the WPF architecture, and encapsulates elements of MVC, MVP, and PM design patterns.

This article will compare and contrast MVC, MVP, and MVVM, and suggest which pattern to use based on your technology of choice and the problem that you are trying to solve.

MVC

A careful reader of the “Gang of Four” book will notice that MVC is not referred to as a design pattern but a “set of classes to build a user interface” that uses design patterns such as Observer, Strategy, and Composite. It also uses Factory Method and Decorator, but the main MVC relationship is defined by the Observer and Strategy patterns.

There are three types of objects. The Model is our application data, the View is a screen, and the Controller defines the way the View reacts to user input. The views and models use the Publish-Subscribe protocol - when Model data is changed, it will update the View. It allows us to attach multiple Views to the same Model [2]. This is achieved by using the Observer design pattern.

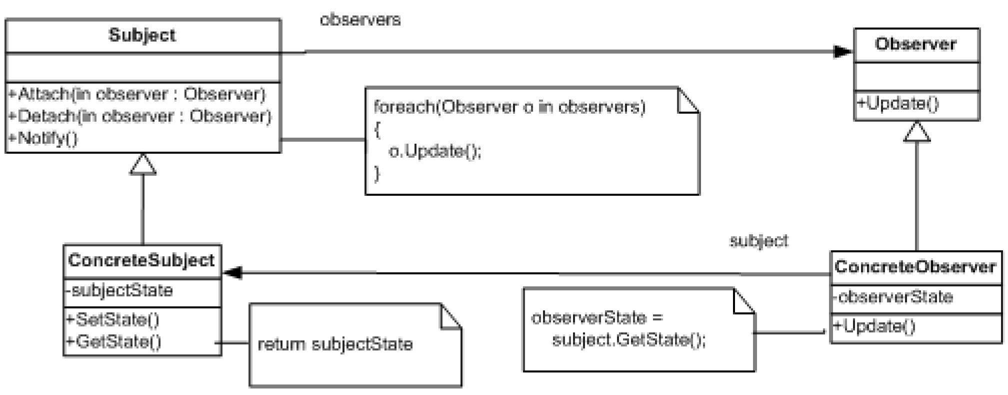

Figure 1: Observer

The goal of this pattern is to define the one-to-many relationship between the Subject and the Observers; if the Subject is changed all Observers are updated. The Subject maintains the list of the Observers and can attach and detach objects to the list. The Observer in return exposes an `Update` method on its interface that Subject can use to update all objects it observes. The C# implementation would look like this:

C# Shrink ▲

```
public abstract class Subject
{
    private readonly ICollection<Observer> Observers =
            new Collection<Observer>();

    public void Attach(Observer observer)
    {
        Observers.Add(observer);
    }
    public void Detach(Observer observer)
    {
        Observers.Remove(observer);
    }
    public void Notify()
    {
        foreach (Observer o in Observers)
        {
            o.Update();
        }
    }
}

public class ConcreteSubject : Subject
{
    public object SubjectState { get; set; }
}

public abstract class Observer
{
    public abstract void Update();
}

public class ConcreteObserver : Observer
{
    private object ObserverState;
    private ConcreteSubject Subject { get; set; }

    public ConcreteObserver(ConcreteSubject subject)
    {
        Subject = subject;
    }
    public override void Update()
    {
        ObserverState = Subject.SubjectState;
    }
}
```

The other component of the MVC pattern is the View-Controller relationship. The View uses the Controller to implement a specific type of response. The controller can be changed to let the View respond differently to user input. This View-Controller link is an example of the Strategy design pattern.

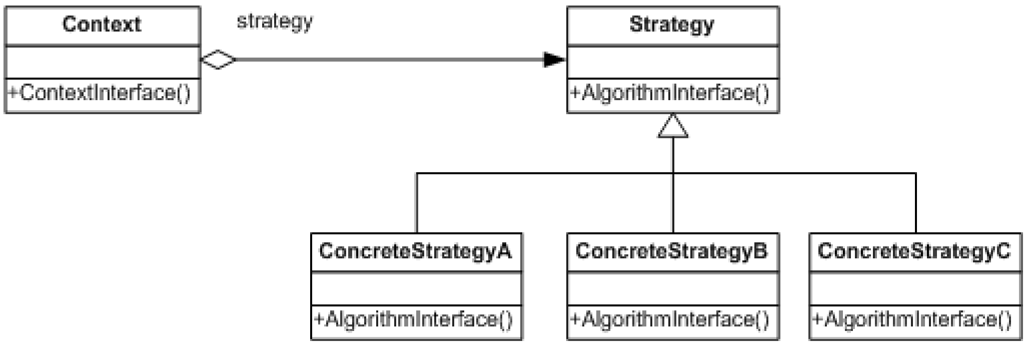

Figure 2: Strategy

Each ConcreteStrategy encapsulates a particular type of response. A Context object has a reference to a Strategy object and can forward the requests to a specific Strategy though the common interface. Here is a C# code:

C#

```
public abstract class Strategy
{
    public abstract void AlgorithmInterface();
}
public class ConcreteStrategyA : Strategy
{
    public override void AlgorithmInterface()
    {
        // code here
    }
}
public class Context
{
    private readonly Strategy Strategy;

    public Context(Strategy strategy)
    {
        Strategy = strategy;
    }
    public void ContextInterface()
    {
        Strategy.AlgorithmInterface();
    }
}
```

Now we know that the Model acts as a Subject from the Observer pattern and the View takes on the role of the Observer object. In the other relationship of the MVC, the View is a Context and the Controller is a Strategy object. Combining our knowledge of the two diagrams, we can draw the MVC UML class diagram as below:

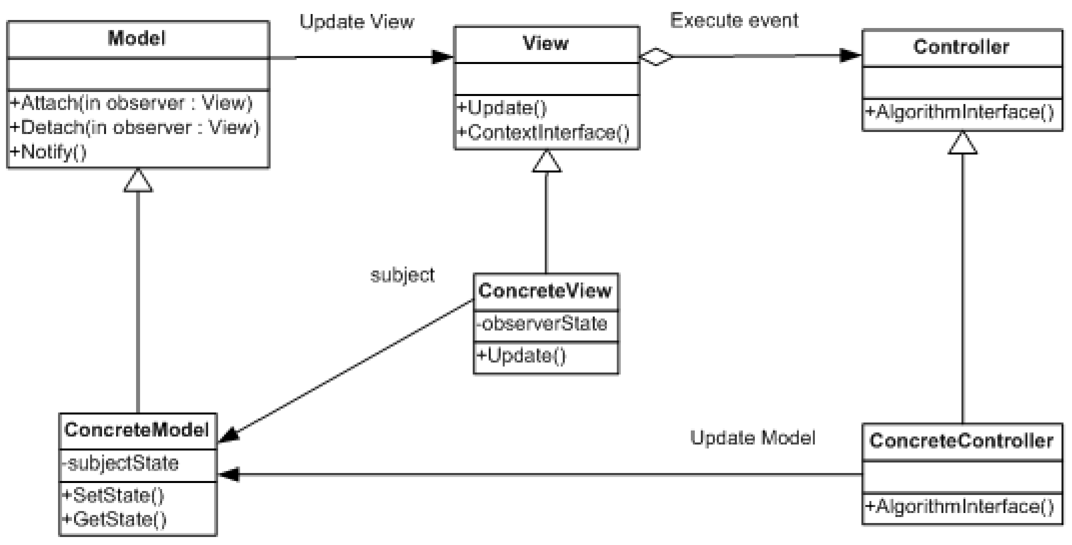

Figure 3: MVC

Here is the implementation code for the MVC pattern:

C#   Shrink ▲

```
public abstract class Model
{
    private readonly ICollection<View> Views = new Collection<View>();
    public void Attach(View view)
    {
        Views.Add(view);
    }
    public void Detach(View view)
    {
        Views.Remove(view);
    }
    public void Notify()
    {
        foreach (View o in Views)
        {
            o.Update();
        }
    }
}

public class ConcreteModel : Model
{
    public object ModelState { get; set; }
}

public abstract class View
{
    public abstract void Update();
    private readonly Controller Controller;
    protected View()
    {
    }
    protected View(Controller controller)
    {
        Controller = controller;
    }
    public void ContextInterface()
    {
        Controller.AlgorithmInterface();
    }
}

public class ConcreteView : View
{
    private object ViewState;
    private ConcreteModel Model { get; set; }
    public ConcreteView(ConcreteModel model)
    {
        Model = model;
    }
    public override void Update()
    {
        ViewState = Model.ModelState;
    }
}

public abstract class Controller
{
    public abstract void AlgorithmInterface();
}

public class ConcreteController : Controller
{
    public override void AlgorithmInterface()
    {
        // code here
    }
}
```

If we leave out the concrete classes for simplicity, we will get a more familiar MVC diagram. Please note that we use pseudo- rather than proper UML shapes, where circles represent a group of classes (e.g., Model and ConcreteModel), not classes as on the UML class diagram on Figure 3.

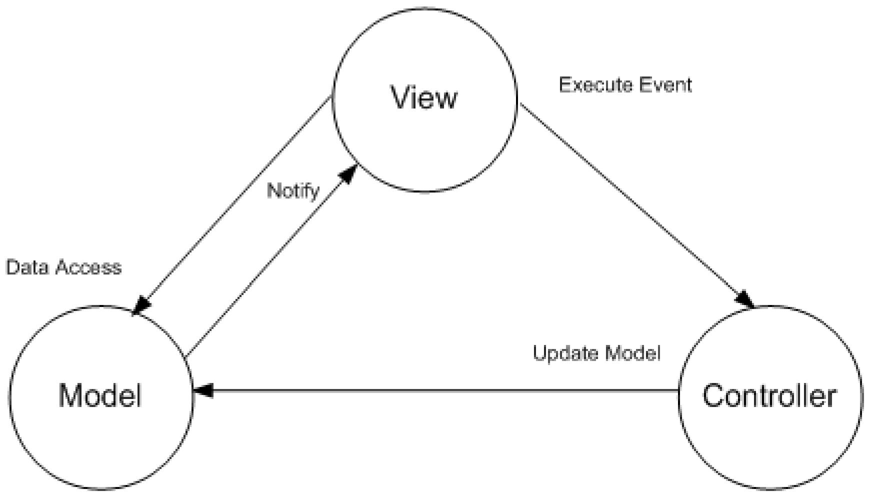

Figure 4: Simplistic MVC

MVP

MVP was first described by Mike Potel from Taligent (IBM) in 1996. Potel in his work on MVP [3] questioned the need for the Controller class in MVC. He noticed that modern Operating System user interfaces already provide most of the Controller functionality in the View class and therefore the Controller seems a bit redundant.

Potel in his paper analyses the types of interactions the View can have with the Model. He classified user actions as selection, execution of commands, and raising events. He, therefore, defined the Selection and Command classes that, as the names suggest, can select a subsection of the Model and perform operations, and also introduced the Interactor class that encapsulates the events that change the data. The new class called the Presenter encapsulates the Selection, Command, and Interactor.

As the MVP evolved, the group of developers working on the Dolphin MVP framework outlined their version in the paper by Andy Bower and Blair McGlashan [4]. Their version is similar to the Potel MVP, but also reviewed the Model-View relationship.

As we have seen, the Model-View relationship is an indirect one based on the Observer design pattern. The Model can notify the View that new data has arrived, and the View can update its data from the Model it is subscribed to. Bower and McGlashan questioned the nature of this indirect link, and suggested that the Model can gain access to the user interface directly.

The basic idea was “twisting MVC” in a way where the View absorbs the Controller functionality and the new class (the Presenter) is added. The Presenter can access the View and the Model directly, and the Model-View relationship can still exist where relevant. Overall, the View displays data and the Presenter can update the model and the view directly.

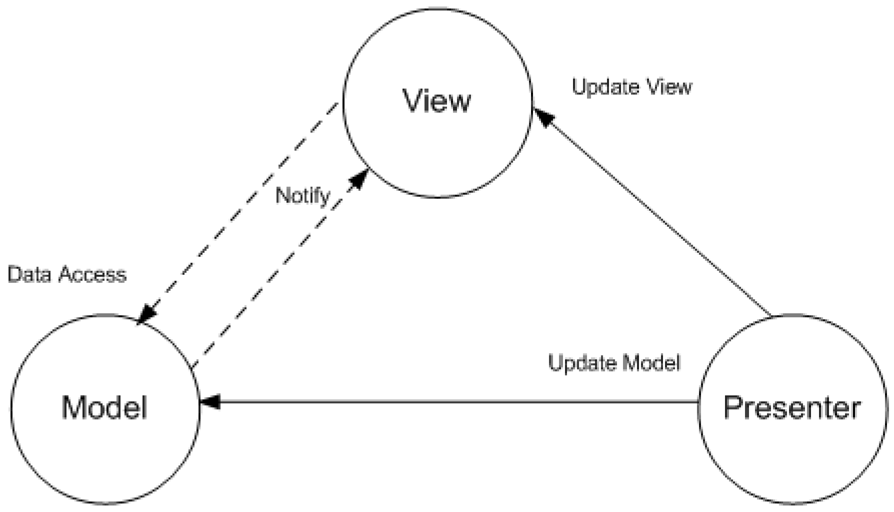

Figure 5: Simplistic MVP

If we add Interactor, Selection, and Command classes, we will get the full picture. There are different flavours of MVP, as we have seen the Presenter can access the View directly, or the Model-View relationship based on the Observer pattern can still exist. In this version, we leave out the Model-View link that we have seen in our MVC code example, and assume that the Presenter can update the View directly.

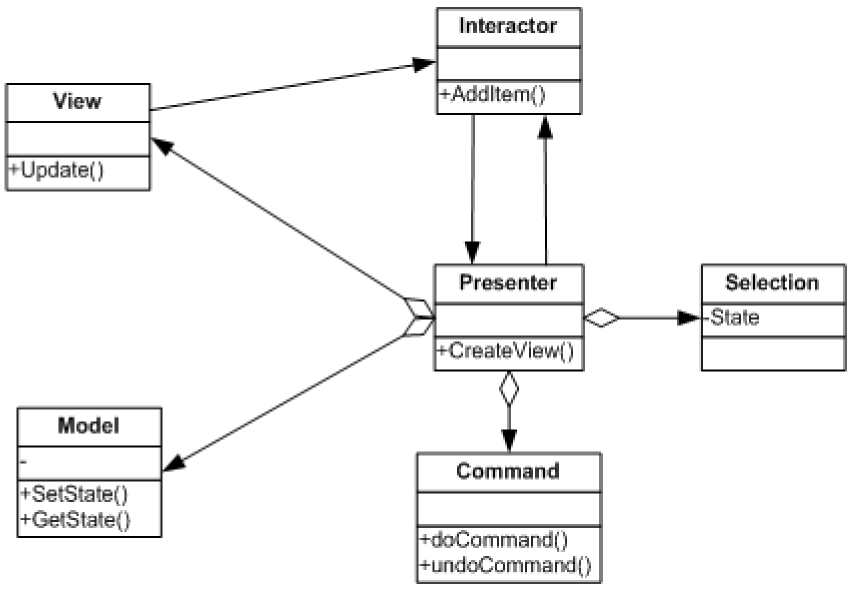

Figure 6: MVP

Here is the implementation code:

C# Shrink ▲

```csharp
public class Model
{
    public object ModelState { get; set; }
}

public class View
{
    private object ViewState;
    // user clicks a button
    public static void Main()
    {
        Interactor interactor = new Interactor();
        Presenter presenter = new Presenter(interactor);
        interactor.AddItem("Message from the UI");
    }
    public void Update(object state)
    {
        ViewState = state;
    }
}

public class Presenter
{
    private readonly Command Command;
    private readonly Interactor Interactor;
    private readonly Model Model;
    private readonly View View;
    public Presenter(Interactor interactor)
    {
        Command = new Command();
        Model = new Model();
        View = new View();
        Interactor = interactor;
        Interactor.ItemAdded +=
          new Interactor.AddItemEventHandler(InteractorItemAdded);
    }
    private void InteractorItemAdded(object sender, Selection e)
    {
        // call command
        Command.DoCommand();
        // update Model
        Model.ModelState = e.State;
        // some processing of the message here
        // ...
        // update View
        View.Update("Processed " + e.State);
    }
}

public class Selection : EventArgs
{
    public object State { get; set; }
}

public class Command
{
    public void DoCommand()
    {
        // code here
    }
}

public class Interactor
{
    public delegate void AddItemEventHandler(object sender, Selection e);
    public event AddItemEventHandler ItemAdded;
    protected virtual void OnItemAdded(Selection e)
    {
        if (ItemAdded != null)
            ItemAdded(this, e);
    }
    public void AddItem(object value)
    {
        Selection selection = new Selection {State = value};
        OnItemAdded(selection);
    }
}
```

Presentation Model

Martin Fowler in his work on GUI architectures [5] not only analysed the UI design patterns in great depths, but also rebranded some of the older patterns, giving them a new life in the modern development frameworks.

The most fundamental idea of MVC is a separation of the presentation objects and the domain model objects. Fowler calls this a Separated Presentation pattern. Separated Presentation gave us the Model and the View. The particular flavour of the Observer pattern they use to communicate, he calls Observer Synchronization, which is different from Flow Synchronization [5]. Therefore, according to Fowler, the Model and the View, and their indirect link, are the core elements that define MVC. The elements that he calls - Separated Presentation and Observer Synchronization.

He also points us towards two quite different descriptions of MVP – one by Potel, and the other one by Bower and McGlashan. Potel was only concerned with removal of the Controller and delegating more work to the View. Fowler calls this approach - Supervising Controller. The other feature described by Bower and McGlashan, the ability of the Presenter to update the View directly, he calls - Passive View.

By breaking down these patterns into smaller pieces, Fowler gives developers the tools they can use when they aim to implement a specific behaviour, not the MVC or MVP patterns themselves. The developers can choose from Separated Presentation, Observer Synchronization, Supervising Controller, or Passive View.

As powerful as they are, both MVC and MVP have their problems. One of them is persistence of the View’s state. For instance, if the Model, being a domain object, does not know anything about the UI, and the View does not implement any business logic, then where would we store the state of the View’s elements such as selected items? Fowler comes up with a solution in the form of a Presentation Model pattern. He acknowledges the roots of the pattern in the Application Model – the fruit of labour of the Visual Works Smalltalk team. Application Model introduces another class between the Model and the View that can store state. For the View, the ApplicationModel class becomes its Model, and the domain specific Model interacts with the ApplicationModel which becomes its View. The ApplicationModel knows how to update the UI, but does not reference any UI elements directly.

Just like in Smalltalk's Application Model, the Presentation Model class sits between the Model and the View. It enriches the Model's data with information such as state that gets synchronised with the View using Publish – Subscribe or the data binding mechanism. The View raises an event that updates the state in the Presentation Model and updates its state in return.

This model allows you to implement the synchronisation code in the View or the Presentation Model. Therefore, it is a developer's decision whether the View should reference the Presentation Model, or the Presentation Model should reference the View [6].

We can draw the diagram below:

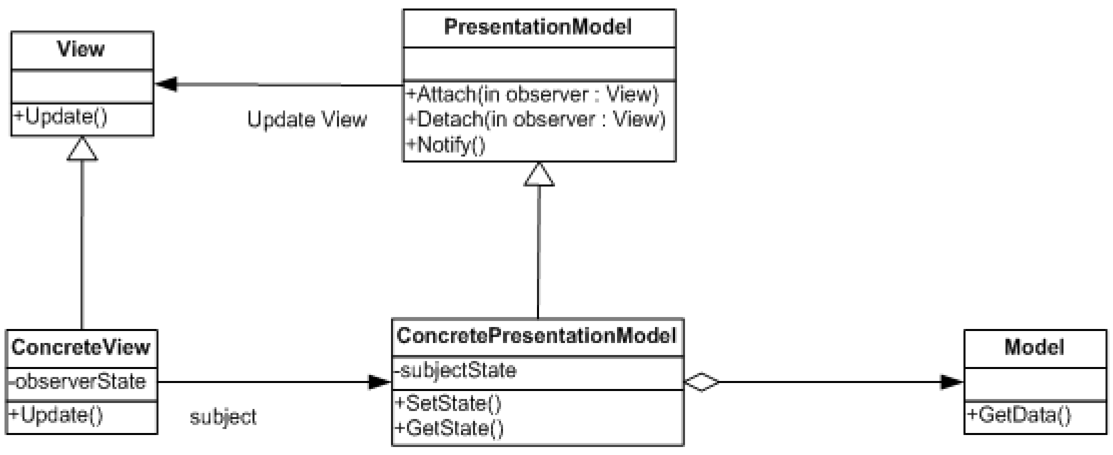

Figure 7: Presentation Model

Here is the code:

C# Shrink ▲

```
public abstract class PresentationModel
{
    private readonly ICollection<View> Views = new Collection<View>();
    public void Attach(View view)
    {
        Views.Add(view);
    }
    public void Detach(View view)
    {
        Views.Remove(view);
    }
    public void Notify()
    {
        foreach (View o in Views)
        {
            o.Update();
        }
    }
}

public class ConcretePresentationModel : PresentationModel
```

```
{
    public Model Model = new Model();
    public object ModelState { get; set; }
}


public class Model
{
    public void GetData()
    {

    }
}

public abstract class View
{
    public abstract void Update();
}

public class ConcreteView : View
{
    private object ViewState;
    private ConcretePresentationModel Model { get; set; }
    public ConcreteView(ConcretePresentationModel model)
    {
        Model = model;
    }
    public override void Update()
    {
        ViewState = Model.ModelState;
    }
}
```

MVVM

The term MVVM was first mentioned by the WPF Architect, John Gossman, on his blog in 2005 [7]. It was then described in depths by Josh Smith in his MSDN article “WPF Apps with the Model-View-ViewModel Design Pattern” [8].

Gossman explains that the idea of MVVM was built around modern UI architecture where the View is the responsibly of the designer rather than the developer and therefore contains no code. Just like its MVC predecessor, the View in MVVM can bind to the data and display updates, but without any coding at all, just using XAML markup extensions. This way, the View is under the designer’s control, but can update its state from the domain classes using the WPF binding mechanism. This fits the description of the Presentation Model pattern.

This is why MVVM is similar to PM where the View will reference the Presentation Model, called ViewModel, which is a better name since it is a “Model of the View”. Unlike the Presentation Model, the ViewModel in MVVM also encapsulates commands just like the Presenter in MVP.

Overall, the View builds the UI and binds to the ViewModel. See code below:

XML

```
< Window x:Class="WpfApplication1.View"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="View" Height="300" Width="300">
    < UniformGrid Columns="2">
        < TextBlock Text="{Binding State}" />
        < Button Command="{Binding GetStateCommand}">Update< /Button>
    < /UniformGrid>
< /Window>
```

The C# code:

C#

```
public partial class View : Window
{
    public View()
    {
        ViewModel viewModel = new ViewModel();
        DataContext = viewModel;
        InitializeComponent();
    }
}
```

The ViewModel provides data such as state, and contains the commands. It also interacts with the Model that provides domain specific objects.

C# Shrink ▲

```
public class ViewModel : INotifyPropertyChanged
{
    private ICommand command;
    private string state;
    private Model model = new Model();

    public string State
    {
        get { return state; }
        set
        {
            state = value;
            OnPropertyChanged("State");
        }
    }

    public ICommand GetStateCommand
    {
        get
        {
            if (command == null)
            {
                command = new RelayCommand(param => DoCommand(),
                                           param => CanDoCommand);
            }
            return command;
        }
        private set { command = value; }
    }
    private void DoCommand()
    {
        State = model.GetData();
    }


    private bool CanDoCommand
    {
        get { return model != null;}
    }
    #region INotifyPropertyChanged Members
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string propertyName)
    {
        var propertyChangedEventArgs =
            new PropertyChangedEventArgs(propertyName);
        if (PropertyChanged != null)
        {
            PropertyChanged(this, propertyChangedEventArgs);
        }
    }
    #endregion
}

/// <summary>
/// http://msdn.microsoft.com/en-us/magazine/dd419663.aspx
/// A command whose sole purpose is to
/// relay its functionality to other
/// objects by invoking delegates.
/// </summary>
public sealed class RelayCommand : ICommand
{
    #region Fields
    readonly Action<object>        m_execute;
    readonly Predicate<object>     m_canExecute;
    #endregion // Fields
    #region Constructors
    /// <summary>
    /// Creates a new command that can always execute.
    /// </summary>
    /// <param name="execute">The execution logic.</param>

    public RelayCommand(Action<object>  execute)
        : this(execute, null)
    { }
    /// <summary>
    /// Creates a new command.
    /// </summary>
    /// <param name="execute">The execution logic.</param>
    /// <param name="canExecute">The execution status logic.</param>

    public RelayCommand(Action<object>  execute, Predicate<object> canExecute)
    {
        if (execute == null)
            throw new ArgumentNullException("execute");
        m_execute = execute;
        m_canExecute = canExecute;
    }
    #endregion // Constructors
    #region ICommand Members
    [DebuggerStepThrough]
    public bool CanExecute(object parameter)
    {
        return m_canExecute == null ? true : m_canExecute(parameter);
    }
    public event EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }
    public void Execute(object parameter)
    {
        m_execute(parameter);
    }
    #endregion // ICommand Members
}
public class Model
{
    public string GetData()
    {
        return "DataFromtheModel";
    }
}
```

This pattern was widely adopted by WPF developers, and can be seen on the UML diagram below:

Summary

As we have seen, the fundamental idea of MVC is a separation of the domain logic and the GUI objects into the Model and the View. These two are linked indirectly by using the Publish-Subscribe mechanism known as the Observer pattern. Another element of this design is a Controller that implements a particular strategy for the View.

It makes sense to use this pattern when the View is very simple and contains no code, as in the case of web based systems and HTML. This way, the View can build the UI, and the Controller handles the user interactions. Note that Microsoft chose to use this pattern in their ASP.NET MVC framework that targets the web developers.

MVP delegates more work to the View and removes the Controller. It introduces the Presenter class that encapsulates the View's state and commands. The Presenter can access the View directly. This is perfect for Windows based systems where Views are powerful classes that can encapsulate the events and store items' state. Note that Microsoft made this a part of the Composite Application Blocks (CAB) framework that targets Windows developers.

The Model – View relationship based on the Observer pattern still exists in MVP; however, the Presenter can access the View directly. Although this is easy to use and implement, developers need to be careful not to break the logic of the system they are trying to model. For instance, the system where the driver accelerates and checks the speed indicator is unlikely to be modeled using MVP. The Driver (Presenter) can update the engine's state (Model), but now needs to update the speed indicator (View). Surely, more logical is the system where the driver updates the engine's state (press gas) and the speed indicator will read these changes (the Observer pattern).

The MVVM is a powerful pattern for all WPF developers. It allows to keep the View free from any code and available to XAML designers. The WPF binding serves as a link to the ViewModel that can handle state, implement the commands, and communicate with the domain specific Model.

MVVM resembles Fowler's PM where the Presentation Model class is called ViewModel.

The Presentation Model is a more general version of MVVM where the developers need to implement the link between the View and the ApplicationModel (ViewModel) themselves. It also allows the View to reference the ApplicationModel, or the ApplicationModel to reference the view. It can be broken down into smaller patterns like Supervising Controller or Passive View where only a specific behaviour is required.

Overall, I hope this paper has achieved its goal to unveil the main GUI patterns and highlight their strengths and weaknesses. It can serve as a good starting point and help you decide what pattern to use when setting out on your next GUI development quest.

References

1. MVC: XEROX PARC 1978-79 (1979) by Trygve Reenskaug, http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html
2. Design Patterns: Elements of Reusable Object-Oriented Software (1994) by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, ISBN 0-201-63361-2
3. MVP: Model-View-Presenter: The Taligent Programming Model for C++ and Java (1996) by Mike Potel, http://www.wildcrest.com/Potel/Portfolio/mvp.pdf
4. Twisting the Triad by Andy Bower, Blair McGlashan, http://www.object-arts.com/papers/TwistingTheTriad.PDF
5. GUI Architectures by Martin Fowler, http://martinfowler.com/eaaDev/uiArchs.html
6. Presentation Model (2004) by Martin Fowler, http://www.martinfowler.com/eaaDev/PresentationModel.html
7. Introduction to Model/View/ViewModel pattern for building WPF apps (2005) by John Gossman, http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx
8. WPF Apps With the Model-View-ViewModel Design Pattern (2009) by Josh Smith, http://msdn.microsoft.com/en-us/magazine/dd419663.aspx