Tidying the House: The MVPC Software Design Pattern

Extending the MVP pattern to abstract the objectives of a presentation from the detail of implementation using modern presentation frameworks

Introduction

This paper provides an overview of the well-known Model / View / Controller (MVC) and Model / View / Presenter (MVP) software design patterns. It then proposes a third pattern, Model / View / Presenter / Controller (MVPC), which extends MVP to abstract the logic associated with the objectives of a presentation from the logic related to the presentation's implementation using modern presentation frameworks.

Model View Controller

Model / View / Controller (MVC) is a software design pattern that divides the application data, screen presentation and user interaction control logic of a system into three separate components.

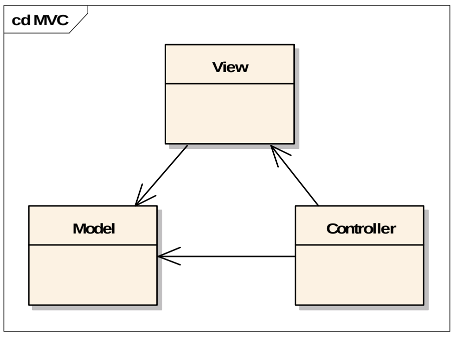


Figure 1: The MVC pattern

The **model** component represents the object, or set of objects, in the application domain. Commonly, we may think of these as the program objects that we create in our software systems to represent logical entities in the business domain. More broadly, the model constitutes objects that embody the purpose of the system in terms of data and behaviour, as opposed to objects that exist purely as framework components that facilitate the operation of the system.

The **view** component presents data from the model on-screen for the user to consume and interact with. Examples of views might be rich Graphical User Interface (GUI) forms or web pages which present data for users to see and manipulate.

In MVC, the view is responsible for ensuring that its presentation reflects the state of the model at all times; when data in the model changes, the view must adjust its presentation to suit. This is often realised by an implementation of the *observer* design pattern, in which each view registers an interest in (or *subscribes* to) changes to the model. When the model is altered, it informs the views registered with it that this has happened, which in turn allows each view to update itself appropriately.

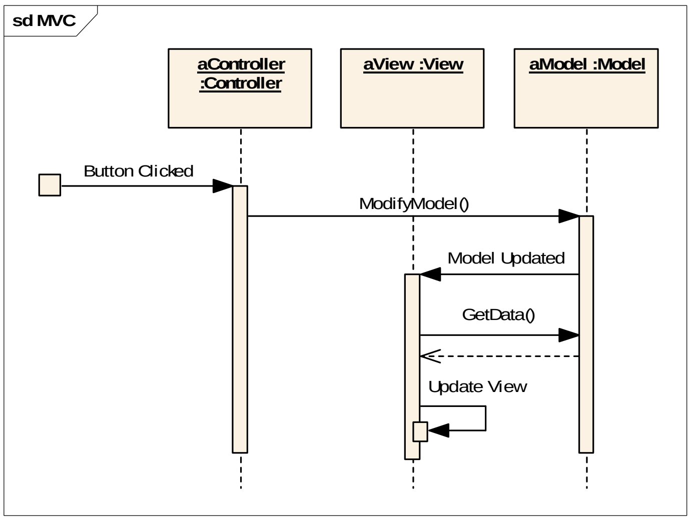


Figure 2: MVC interactions

The **controller** component defines and manages the way the view reacts to input from the user. When a given event "happens" to a view, it is the controller that catches the event and decides what happens next.

Model View Presenter

Through the mechanism outlined above, the MVC pattern *decouples* the model from views, and views from the logic that controls them. This allows each component to be modified independently, with minimal impact on each other. Views can be added and modified without impacting the model; view control logic can be altered without impacting the view(s) to which it relates. MVC, then, embodies software design best practices in that it encourages the separation of concerns and reduces dependency between components.

All this is great. However, there is a small catch. Nowadays, most of the common rich-client presentation technologies (such as Java Swing and WinForms in Microsoft .NET) lean heavily towards embedding presentation logic within views. View classes are often composed of controls, are studded with code to catch and react to events raised by these controls, and embed logic to alter the state of controls when required. As a result, view classes can become overly complex and cumbersome to develop, test and maintain.

The Model / View / Presenter (MVP) design pattern separates the view from its presentation logic to allow each to vary independently. In MVP, the view becomes an ultra-thin component whose purpose is purely to provide a presentation to the user. The view catches and handles events raised by the user, but forwards these directly to the presenter who knows how to deal with them. The presenter communicates with the model, and coordinates directly with the view's controls to present data.

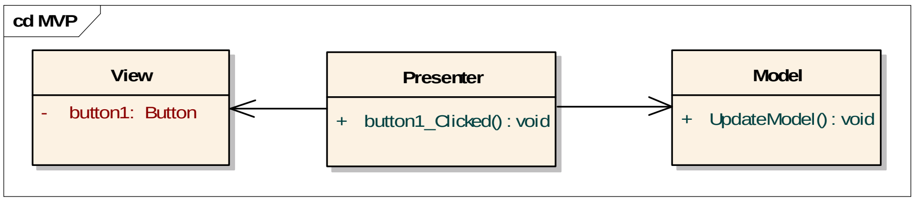


Figure 3: The MVP pattern

The relationship between view and presenter in MVP roughly follows the *decorator* design pattern. The presenter *decorates* the view with presentation logic, since all of the operations in the view which would otherwise deal with presentation logic themselves, instead delegate this responsibility to the presenter.

In separating the view and presentation logic, MVP simplifies both. As a result, the entire presentation is much less complex and cumbersome to develop and maintain. Importantly, MVP facilitates unit testing of presentation logic without the need for human involvement in those tests, and also promotes reuse by allowing *different* actual views to be constructed for a single presenter (for example a rich-client form and a web-form).

MVP is very similar to MVC. However, with MVP the view catches events raised and *forwards* them to the controller (presenter). With MVC, the controller catches events raised directly. It is by virtue of the way that today's common presentation technologies have moved to embed presentation logic *within* a view that MVP is required.

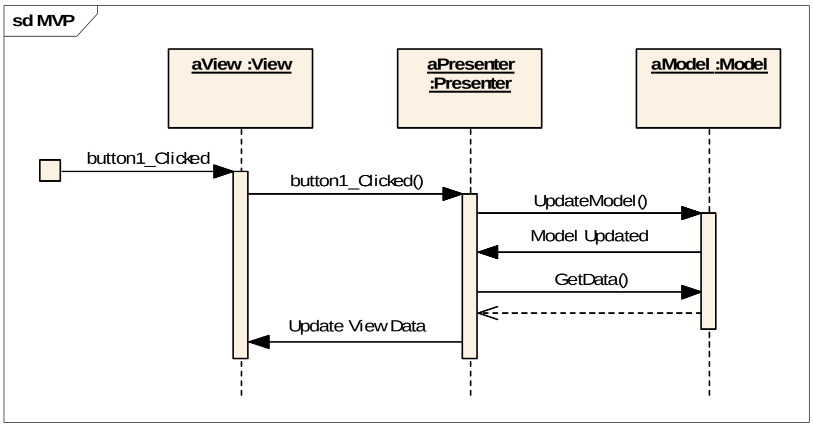


Figure 4: MVP interactions

Model View Presenter Controller

Neither the MVC nor the MVP pattern attempts to separate the material aspects of a presentation from its abstract purpose. For example, in both MVC and MVP, events raised by controls on a view are caught and dealt with by code within a controller. But the higher order code relating to the abstract purpose of the presentation is wrapped up tightly with lower order code relating to specifics of the proprietary presentation framework, like events and controls. As such, the controls, events and other logical elements of a presentation's view cannot be varied independently of the logic relating to the purpose of the presentation.

Imagine a view which presents a limited vocabulary of calendar months using a select box control. When the selected month is modified by the user, the select box control raises an event which is then caught either directly (in MVC) or indirectly (in MVP) by the controller. The controller's event handling code then executes some logic, related to the abstract purpose of the presentation, which may result in the model data being changed in some way.

Now let us propose that you wanted to change the way in which the limited vocabulary was presented to use instead a set of radio buttons. You would need to remove the presentation logic associated with the previous (select box) control and event from the controller, and re-code similar logic for the radio button control and event instead. Even though the actual code you might author in the controller may be subtly different in each case, the abstract purpose of the code would be identical.

With MVC and MVP, then, the abstract purpose of a presentation is inextricably tied to the specifics of the framework of forms, controls and events that deal with its actual rendering and behaviour on-screen.

So, how can we separate the concerns of presentation-purpose related code from presentation-view-implementation code so that they can both vary without impacting one another? What we need is a model in which the view-related and purpose-related logic in the presenter are abstracted from one another and can be varied independently.

The following pattern, called Model / View / Presenter / Controller (MVPC), embodies such principles. With MVPC, the presenter component of the MVP model is split into two components: the *presenter* (view control logic) and *controller* (abstract purpose control logic).

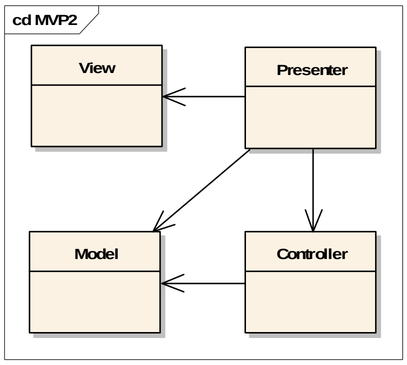


Figure 5: The MVPC pattern

In the MVPC model, the view, presenter and controller are separated so that each can vary independently. The model abstracts the view control logic from the abstract purpose logic. This allows view control logic to be varied without impacting abstract purpose logic, and vice versa.

Imagine a scenario in which two events triggered by controls on a view have the same abstract purpose for the presentation. An example might be the click event of a submit button and the enter-key-press event on a text field, where both events might trigger behaviour in the presentation controller to submit view data to the model. Now, any developer worth their salt would ordinarily functionally decompose the logic of these events such that the event handler code for both events would call a single internal operation which embeds logic to realise the behaviour. All we are doing in MVPC is abstracting the internal operation out of the presenter and putting it into another class, the controller. This might seem like a simple change, but it offers significant benefits over the MVP model. Separation of logic means that the presenter and controller can be interchanged with little impact on each other.

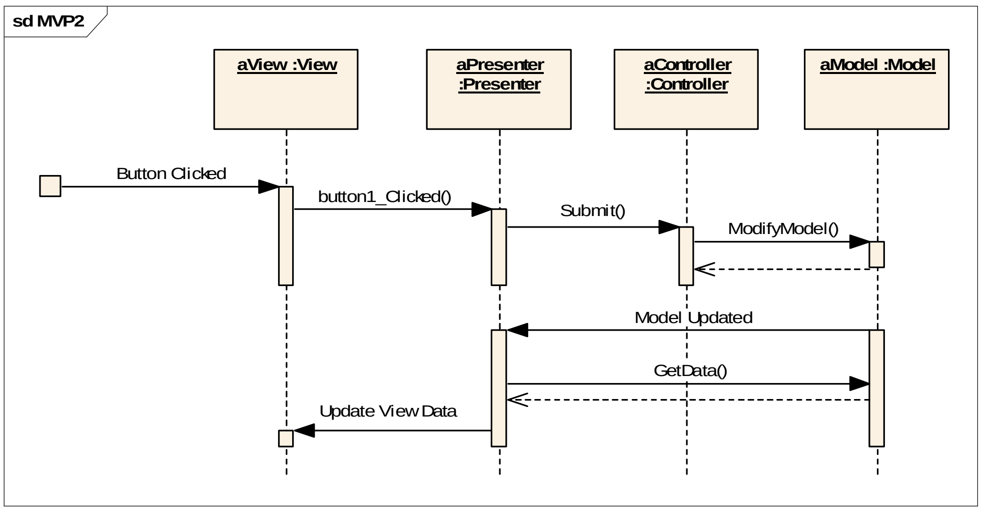


Figure 6: MVPC interactions

We can improve separation further by abstracting the interface of the controller into an interface type which is realised by the controller.

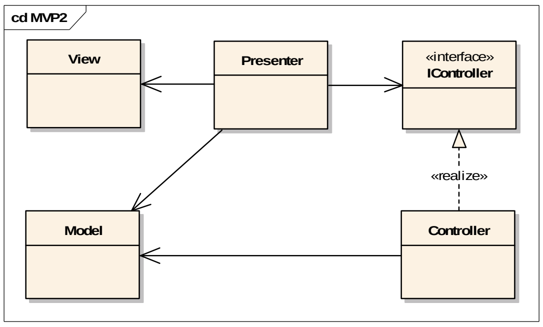


Figure 7: MVPC with abstracted Controller interface

The MVPC pattern improves the MVP pattern as follows:

1. Controllers applying different control logics can be interchanged without impacting the presenter or view.

2. Completely different views (utilising, say, a different technology) can be applied to the same controller without impacting the underlying control logic.

3. Multiple related views can be coordinated by a single controller. An example might be in the implementation of a wizard, where multiple views need to be coordinated by a single set of control logic.

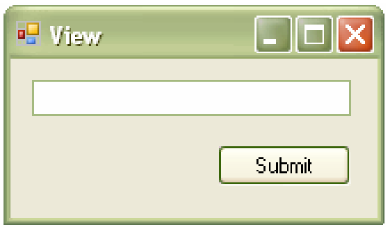
The application of the bridge and / or factory design patterns to the presenter-controller relationship can further decouple the two and improve the extensibility of the MVPC pattern. By applying the *bridge*, the purpose of a controller can be abstracted from its implementation such that the two can vary independently. By applying the *factory*, the creational aspects of the controller can be encapsulated in a separate class so that the presentation doesn't need to know what is implementing the control logic specifically.

Coding the MVPC pattern

So, let's look at an example of the MVPC pattern in Microsoft C# program code. In the code sample on the following page, the interface class *IController* defines operations for the control logic relating to the purpose of the presentation. The class *Controller* implements this control logic. The class *Presenter* deals specifically with the presentation logic tasks associated with the View's controls, events and so on. The *View* is coordinated by the Presenter to present data to the user.

For brevity we have substituted the *Model* in the code sample with comments in the `Submit()` operation of the Controller. In practice, the model would represent the application domain as with MVP and MVC, and the `Submit()` Controller operation would be doing much more!

In Microsoft C#, the View class would present on-screen as follows. Note that the View form comprises two controls, a textbox (blank) and a button (marked "Submit"):



The View class

```
public partial class View : Form
{
    private Presenter _pres;

    public View()
    {
        _pres = new Presenter();
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        _pres.button1_Click();
    }

    private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
    {
        _pres.textBox1_KeyPress();
    }
}
```

The Presenter class

```
public class Presenter
{
    private IController _cont;

    public Presenter()
    {
        _cont = new Controller();
    }

    public void button1_Click()
    {
        _cont.Submit();
    }

    public void textBox1_KeyPress()
    {
       // Note: Only do this if key code is “enter”
        _cont.Submit();
    }
}
```

The IController interface and Controller class

```
public interface IController
{
    void Submit();
}

public class Controller : IController
{
    public void Submit()
    {
        // Do something to the Model here!
    }
}
```