Joel in point form

MVVM vs MVP vs MVC: The differences explained

MAY 6, 2011 BY JOEL 21 COMMENTS POSTED UNDER: LINKED IN, SOFTWARE DEVELOPMENT

For the consicise explanation, see MVVM vs MVP vs MVC: The concise explanation

Those who know me know that I have a passion for software architecture and after developing projects using Model-View-ViewModel (MVVM), Model-View-Presenter (MVP), and Model-View-Controller (MVC), I finally feel qualified to talk about the differences between these architectures. The goal of this article is to clearly explain the differences between these 3 architectures.

Contents [hide]

Common Elements

First, the let's define common elements. All 3 of the architectures are designed to separate the view from the model.

Model

Domain entities & functionality

Knows only about itself and not about views, controllers, etc.

For some projects, it is simply a database and a simple DAO

For some projects, it could be a database/file system, a set of entities, and a number of classes/libraries that provide additional logic to the entities (such as performing calculations, managing state, etc)

Implementation: Create classes that describe your domain and handle functionality. You probably should end up with a set of domain objects and a set of classes that manipulate those objects.

View

Code that handles the display

Note that view related code in the codebehind is allowed (see final notes at the bottom for details)

Implementation: HTML, WPF, WindowsForms, views created programmatically – basically code that deals with display only.

Differences between Presenters, ViewModels and Controllers

This is the tricky part. Some things that Controllers, Presenters, and ViewModels have in common are:

RELATED (DESIGN)
Solving Traveling Salesman with Ant Colony Optimization in JavaScript
MVVM vs MVP vs MVC: The concise explanation
Explanation of the stack, the heap, and recursion leading to stack overflow
Polymorphism in C#: new vs override as function modifiers
Garbage Collection FAQ
3 Rules of SDK Design
MVVM vs MVP vs MVC: The differences explained
Code Maintenance: Avoiding Special Cases

POPULAR POSTS
MVVM vs MVP vs MVC: The differences explained
Burn Down Chart Tutorial: Simple Agile Task Tracking
Algorithms and Data Structure Interview Preparation Guide
Solving Traveling Salesman with Ant Colony Optimization in JavaScript
.Net Interview Questions 7: WPF
MVVM vs MVP vs MVC: The concise explanation
Interview Questions for Software Developers
.Net Interview Questions 3: Framework Core
Explanation of the stack, the heap, and recursion leading to stack overflow
3 Rules of SDK Design
.Net Interview Questions 4: Multithreading
Pyramid js: A dependency manager to load/combine web development files

ABOUT

Thin layers

They communicate with the model and the view

I am a Toronto based software engineer currently working for Amazon.

More | **Subscribe (RSS)**

The features of each.

Presenter (Example: WinForms)

2 way communication with the view

View Communication: The view communicates with the presenter by directly calling functions on an instance of the presenter. The presenter communicates with the view by talking to an interface implemented by the view.

There is a single presenter for each view

Implementation:

Every view's codebehind implements some sort of IView interface. This interface has functions like displayErrorMessage(message:String), showCustomers(customers:IList<Customer>), etc. When a function like showCustomers is called in the view, the appropriate items passed are added to the display. The presenter corresponding to the view has a reference to this interface which is passed via the constructor.

In the view's codebehind, an instance of the presenter is referenced. It may be instantiated in the code behind or somewhere else. Events are forwarded to the presenter through the codebehind. The view never passes view related code (such as controls, control event objects, etc) to the presenter.

A code example is shown below.

```
//the view interface that the presenter interacts with
public interface IUserView
{
    void ShowUser(User user);
    ...
}

//the view code behind
public partial class UserForm : Form, IUserView
{
    UserPresenter _presenter;
    public UserForm()
    {
        _presenter = new UserPresenter(this);
        InitializeComponent();
    }

    private void SaveUser_Click(object sender, EventArgs e)
    {
        //get user from form elements
        User user = ...;
        _presenter.SaveUser(user);
    }

    ...
}

public class UserPresenter
{
    IUserView _view;
    public UserPresenter(IUserView view){
        _view = view;
    }

    public void SaveUser(User user)
    {
        ...
    }
    ...
}
```

ViewModel (Example: WPF, Knockoutjs)

2 way communication with the view

The ViewModel represents the view. This means that fields in a view model usually match up more closely with the view than with the model.

View Communication: There is no IView reference in the ViewModel. Instead, the view binds directly to the ViewModel. Because of the binding, changes in the view are automatically reflected in the ViewModel and changes in the ViewModel are automatically reflected in the view.

There is a single ViewModel for each view

Implementation:

The view’s datacontext is set to the ViewModel. The controls in the view are bound to various members of the ViewModel.

Exposed ViewModel proproperties implement some sort of observable interface that can be used to automatically update the view (With WPF this is INotifyPropertyChanged; with knockoutjs this is done with the functions ko.observable() and ko.observrableCollection())

Controller (Example: ASP.NET MVC Website)

The controller determines which view is displayed

Events in the view trigger actions that the controller can use to modify the model or choose the next view.

There could be multiple views for each controller

View Communication:

The controller has a method that determines which view gets displayed

The view sends input events to the controller via a callback or registered handler. In the case of a website, the view sends events to the controller via a url that gets routed to the appropriate controller and controller method.

The view receives updates directly from the model without having to go through the controller.

Note: In practice, I don’t think this particular feature of MVC is employed as often today as it was in the past. Today, I think developers are opting for MVVM (or MVP) over MVC in most situations where this feature of MVC would have been used. Websites are a situation where I think MVC is still a very practical solution. However, the view is always disconnected from the server model and can only receive updates with a request that gets routed through the controller. The view is not able to receive updates directly from the model.

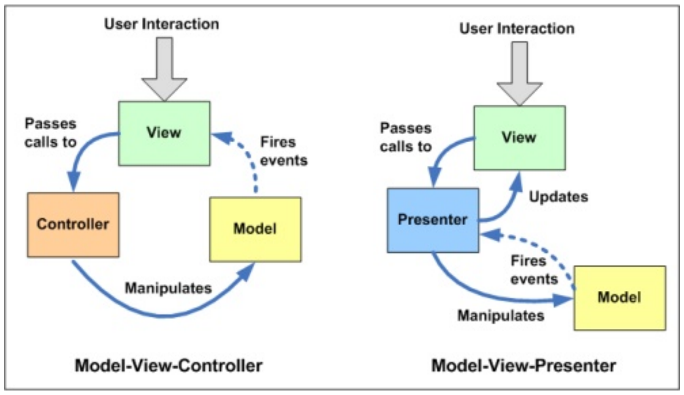
Implementation (for web):

A class is required to interpret incoming requests and direct them to the appropriate controller. This can be done by just parsing the url. Asp.net MVC does it for you.

If required, the controller updates the model based on the request.

If required, the controller chooses the next view based on the request. This means the controller needs to have access to some class that can be used to display the appropriate view. Asp.net MVC provides a function to do this that is available in all controllers. You just need to pass the appropriate view name and data model.

MVVM and MVP implementation seem pretty straightforward but MVC can be a little confusing. The diagram below from Microsoft’s Smart Client Factory documentation does a great job at showing MVC communication. Note that the controller chooses the view (ASP.NET MVC) which is not shown in this diagram. MVVM interactions will look identical to MVP (replace Presenter with ViewModel). The difference is that with MVP, those interactions are handled programmatically while with MVVM, they will be handled automatically by the data bindings.



General rules for when to use which?

MVP

Use in situations where binding via a datacontext is not possible.

Windows Forms is a perfect example of this. In order to separate the view from the model, a presenter is needed. Since the view cannot directly bind to the presenter, information must be passed to it view an interface (IView).

MVVM

Use in situations where binding via a datacontext is possible. Why? The various IView interfaces for each view are removed which means less code to maintain.

Some examples where MVVM is possible include WPF and javascript projects using Knockout.

MVC

Use in situations where the connection between the view and the rest of the program is not always available (and you can't effectively employ MVVM or MVP).

This clearly describes the situation where a web API is separated from the data sent to the client browsers. Microsoft's ASP.NET MVC is a great tool for managing such situations and provides a very clear MVC framework.

Final notes

Don't get stuck on semantics. Many times, one of your systems will not be purely MVP or MVVM or MVC. Don't worry about it. Your goal is not to make an MVP, MVVM, or MVC system. Your goal is to separate the view, the model, and the logic that governs both of them. It doesn't matter if your view binds to your 'Presenter', or if you have a pure Presenter mixed in with a bunch of ViewModels. The goal of a maintainable project is still achieved.

Some evangelists will say that your ViewModels (and Presenters) must not make your model entities directly available for binding to the view. There are definitely situations where this is a bad thing. However, don't avoid this for the sake of avoiding it. Otherwise, you will have to constantly be copying data between your model and ViewModel. Usually this is a pointless waste of time that results in much more code to maintain.

In line with the last point, if using WPF it makes sense to implement INotifyPropertyChanged in your model entities. Yes, this does break POCO but when considering that INotifyPropertyChanged adds a lot of functionality with very little maintenance overhead , it is an easy decision to make.

Don't worry about "bending the rules" a bit so long as the main goal of a maintainable program is achieved

Views

When there is markup available for creating views (Xaml, HTML, etc), some evangelists may try to convince developers that views must be written entirely in markup with no code behind. However, there are perfectly acceptable reasons to use the code behind in a view if it is dealing with view related logic. In fact, it is the ideal way to keep view code out of your controllers, view models, and presenters. Examples of situations where you might use the code behind include:

Formatting a display field

Showing only certain details depending on state

Managing view animations

Examples of code that should not be in the view

Sending an entity to the database to be saved

Business logic

Happy coding.

Tags: programming, software design

21 Comments

+ ADD COMMENT

Vikram Purohit September 4, 2011 at 8:11 am

REPLY

Hey joel,

Thanks for the fantastic article. Can we adopt a model which will be same for web and windows by just replacing the presentation Layer?

Joel September 4, 2011 at 12:37 pm

REPLY

Yes, your model is independent of the views and the Controllers/Presenters/ViewModels so you should always be able to use the same model for multiple view methodologies. If you are planning on having similar functionality in the windows and web apps, you likely will even be able to share the ViewModels/Presenters.

sham parnerkar September 7, 2011 at 6:04 am

REPLY

Nice And Easy to understand article Joel, Clearing almost all doubts about MVC-MVP-MVVM.

Ravi Kumar Gupta January 12, 2012 at 10:59 am

REPLY

I was confuse with the real differences between MVP, MVC and MVVM and your article has cleared my doubts. Thanks Joel!

Ravinatha Reddy March 8, 2012 at 6:57 pm

REPLY

This is a fantastic article, many time developers are confused about various patterns especially in ASP.Net forms. The real distinction between code-behind and controller class is made clear. The bottom line is to keep Model , View and Controller/Presenter/Manager classes thin
and responsible of their concerns.

Bewnet April 14, 2012 at 4:33 am

REPLY

It's really impressive and to the point article. Cheers for sharing it.
I have a little doubt and wanted to ask to get more idea.
First, you mentioned that in MVP, you said, "There is a single presenter for each view", cann't we have to use say more than one presenter for the same View?
My main question is actually on MVC, can you give me an example of a "controller will choose a view" Is this about page redirecting? Sorry if this is silly question but, i want to understand it well.

Thanks for your time and keep posting your good work!

Joel April 16, 2012 at 9:08 pm

REPLY

In MVP, I don't see how you could really have more than one presenter in each view. There are always variations but I'm not sure that I would call that variation MVP. However, you can always have a view that is made up of multiple sub views. Each subview can be separately defined and each will have its own presenter. This situation occurs a lot in MVP. For example, you might have a tab that has a member profile view and a member's recent posts view. In this case, I could see having a custom UserControl for the member profile and a custom UserControl for the recent posts. Each of these could have their own presenters and be combined in the tab view.

As for your MVC question, it sounds a bit like it is about redirecting…however, redirecting would change the URL. What you are really doing is supplying the appropriate content (view html) for a given url. One controller would handle this functionality for a set of related views. Does that answer your question?

Dart January 9, 2014 at 7:57 am

REPLY

An example of different presenters of the view.
Imagine, you have ICustomersView which is used both for adding new customers and editing existing ones. Imagine, that business logic is different for new customer and previously saved one. Some field may become readonly, some actions couldn't be made with fresh new customer, etc. You have 2 ways:
1) Create single presenter, which knows about current edit mode – NewCustomerMode or EditCustomerMode. It will have a lot of "if's to handle this difference.
2) Create 2 presenters – CustomerAddPresenter and CustomerEditPresener, each of them doing his own job. If some functionality is similar for both cases – move it to base presenter class.

I believe that second way is better. Less code in each presenter, easier to create unit-tests.

But that won't work if view creates the presenter in code-behind. I believe, that view have no need to know about presenter at all. Using events declared in IView instead of direct calls allows decoupling view from presenter.

Ralph Krausse June 6, 2012 at 12:42 pm

REPLY

Nice explanation. Thanks!

Ritesh August 13, 2012 at 7:44 am

REPLY

Hi,

This is such a nice, and precise explanation of three patterns. I have a query though and

possibly you can help me. I am developing an Desktop Client and Web application. I intend to use WPF (and MVVM) to develop desktop client and ASP.Net MVC for web app.

I have never used ASP.Net MVC before although I have never truly liked Web Forms (except some features like Master Page, output Cache etc.) and I mostly use

AJAX (jQuery), and handlers to populate HTML and processing inputs (I think I was

close to MVC after reading about the pattern but in a different way).

Now this application will mostly have same inputs, reports, and database. I am planning to create Model that can be re-used in MVVM and MVC both. But after reading your article, analysis of ASP.Net MVC Code, I doubt that it could be done. In MVVM, View never knows Model while in MVC, Controller shares Model with View. Also, in ASP.Net MVC, a View (ASPX file) is derived from System.Web.Mvc.ViewPage and the labels/captions are populated from Model itself.

Is there a way I can use same Model for both applications?

Thank you.

Ritesh

Santhosh Varghese September 2, 2012 at 8:03 am REPLY

Hi Joe,

This is a wonderful,specific topic and not much and not less.
You clearly mentioned the points of use of each pattern.

Thanks a lot..

Bhushan Mulmule October 29, 2012 at 7:11 am REPLY

Excellent explanation. Best thing is examples for each approach.

Neha Khanna November 14, 2012 at 2:23 am REPLY

Thanks,

Now this helped me putting down my final yes on knockoutJS framework for my new HTML5, JavaScript based application.

Jacob Page November 15, 2012 at 7:15 pm REPLY

Nice summary. I disagree with one of the assertions, however. With MVVM, you don't need to have one VM per view. Multiple views can be data bound to the same view model. This is very helpful when you have multiple visualizations for the same core data that need to be kept in-sync.

OlegX November 29, 2012 at 3:58 pm REPLY

Excellent article – simple and very logic, clarifying all three patterns

Akshay January 22, 2013 at 4:55 am REPLY

Very well explained!!…It satisfies most of our doubts about these patterns.

Dov March 7, 2013 at 11:37 am REPLY

Regarding MVC, I think you have a mistake:
MVC is a methodology related to the presentation layer, so it has nothing to do with your Domain model!

The Model in MVC represents a projection of your data, customized so that the View is able to show it. In other words Model in MVC is ViewModel!
For further reading, you can read Dino Esposito's book "Programming Microsoft® ASP.NET MVC 2″. He explains this very well and Conceivably.

Apart this comment – thanks for the article!

Joel September 2, 2013 at 6:20 pm REPLY

Well said. I will try to find a way to update the wording. Domain seems to imply something in the service tier and not the web tier so I will try to find a more appropriate term. One issue is that the term ViewModel is a loaded term with multiple meanings. One, is from the concept of ViewModel as described by MVVM. In this case, the ViewModel responds to changes in the view and forwards them to a model representation that more accurately represents the business concepts (The M that I describe as the domain model). The other way people use the term is when they think of models associated with the web tier (and call them view models). I think this is how you use the term ViewModel but when using it this way, it leaves part of MVVM undefined.

Kirti July 21, 2013 at 4:58 am REPLY

Excellent article on explaining the differences between the 3 patterns !! Thanks..

Harshal Patil January 3, 2014 at 11:57 am REPLY

Hi Joel,

Very nice and concise description. You have cleverly illustrated the difference in all the three patterns. However, the

MVC interaction pattern that you shared has a minor issue. User inputs are not transferred from View to Controller. Rather, controller is the one that directly receives the user inputs which converts them to command for view and models. But, that is just my understanding so far after reading the classic literature on MVC. I may be wrong.

Joel January 26, 2014 at 9:58 pm REPLY

Depending on how you consider system interaction, you could very well be correct here. When thinking of a web based MVC framework, I think like to think I'm interacting with the controllers directly through the view (since by making changes to the view I am influencing the actions in the controller). However, this is a bit of a simplification since I could actually just send the proper HTTP request directly to the server (and controller) without having to go through the view. I personally find thinking of going through the view as a little more clear for my case since I actually interact through the view but I can understand why some people may have different ways of considering this part. This point doesn't seem to have a big overall impact on how I consider these types of frameworks but perhaps my opinion will change after I think on it some more. Thanks for sharing.

Got anything to say? Go ahead and leave a comment!

Name (required)

Mail (will not be published) (required)

Website

Comment

POST COMMENT

Favourite Links

- Best way to find bugs in your code
- Don't call yourself a programmer
- Google Finance
- Our Signal
- Programmer Competency Matrix
- Stack Overflow
- Stock Options Manager
- Wikipedia