What are MVP and MVC and what is the difference?

Ask Question

When looking beyond the RAD (drag-drop and configure) way of building user interfaces that many tools encourage you are likely to come across three design patterns called Model-View-Controller, Model-View-Presenter and Model-View-ViewModel. My question has three parts to it:

1. What issues do these patterns address?
2. How are they similar?
3. How are they different?

design-patterns model-view-controller user-interface mvp glossary

edited May 4 '15 at 3:26

Peter Mortensen
12.3k 18 81 107

asked Aug 5 '08 at 10:06

Mike Minutillo
21.7k 12 40 41

protected by NullPointer Jun 10 '13 at 5:12

This question is protected to prevent "thanks!", "me too!", or spam answers by new users. To answer it, you must have earned at least 10 reputation on this site (the association bonus does not count).

2 mvc.givan.se/#mvp – givanse Dec 11 '16 at 16:51

1 IDK, but supposedly for the original MVC, it was meant to be used in the small. Each button, label, etc, had its' own view and controller object, or at least that is what Uncle Bob claims. I think he was talking about Smalltalk. Look up his talks on YouTube, they are fascinating. – still_dreaming_1 Sep 3 '17 at 1:19

MVP adds an extra layer of indirection by splitting the View-Controller into a View and a Presenter... – the_prole Jan 26 at 2:33

The main difference is that in MVC the Controller does not pass any data from the Model to the View. It just notifies the View to get the data from the Model itself. However, in MVP, there is no connection between the View and Model. The Presenter itself gets any data needed from the Model and passes it to the View to show. More to this together with an android sample in all architecture patterns is here: digigene.com/category/android/android-architecture – Ali Nemati Hayati Mar 19 at 11:49

25 Answers

Model-View-Presenter

In **MVP**, the Presenter contains the UI business logic for the View. All invocations from the View delegate directly to Presenter. The Presenter is also decoupled directly from the View and talks to it through an interface. This is to allow mocking of the View in a unit test. One common attribute of MVP is that there has to be a lot of two-way dispatching. For example, when someone clicks the "Save" button, the event handler delegates to the Presenter's "OnSave" method. Once the save is completed, the Presenter will then call back the View through its interface so that the View can display that the save has completed.

MVP tends to be a very natural pattern for achieving separated presentation in Web Forms. The reason is that the View is always created first by the ASP.NET runtime. You can find out more about both variants.

Two primary variations

Passive View: The View is as dumb as possible and contains almost zero logic. The Presenter is a middle man that talks to the View and the Model. The View and Model are completely shielded from one another. The Model may raise events, but the Presenter subscribes to them for updating the View. In Passive View there is no direct data binding, instead the View exposes setter properties which the Presenter uses to set the data. All state is managed in the Presenter and not the View.

- Pro: maximum testability surface; clean separation of the View and Model
- Con: more work (for example all the setter properties) as you are doing all the data binding yourself.

Supervising Controller: The Presenter handles user gestures. The View binds to the Model directly through data binding. In this case it's the Presenter's job to pass off the Model to the View so that it can bind to it. The Presenter will also contain logic for gestures like pressing a button, navigation, etc.

- Pro: by leveraging databinding the amount of code is reduced.

- Con: there's less testable surface (because of data binding), and there's less encapsulation in the View since it talks directly to the Model.

Model-View-Controller

In the **MVC**, the Controller is responsible for determining which View to display in response to any action including when the application loads. This differs from MVP where actions route through the View to the Presenter. In MVC, every action in the View correlates with a call to a Controller along with an action. In the web each action involves a call to a URL on the other side of which there is a Controller who responds. Once that Controller has completed its processing, it will return the correct View. The sequence continues in that manner throughout the life of the application:

```
Action in the View
    -> Call to Controller
    -> Controller Logic
    -> Controller returns the View.
```

One other big difference about MVC is that the View does not directly bind to the Model. The view simply renders, and is completely stateless. In implementations of MVC the View usually will not have any logic in the code behind. This is contrary to MVP where it is absolutely necessary because, if the View does not delegate to the Presenter, it will never get called.

Presentation Model

One other pattern to look at is the **Presentation Model** pattern. In this pattern there is no Presenter. Instead the View binds directly to a Presentation Model. The Presentation Model is a Model crafted specifically for the View. This means this Model can expose properties that one would never put on a domain model as it would be a violation of separation-of-concerns. In this case, the Presentation Model binds to the domain model, and may subscribe to events coming from that Model. The View then subscribes to events coming from the Presentation Model and updates itself accordingly. The Presentation Model can expose commands which the view uses for invoking actions. The advantage of this approach is that you can essentially remove the code-behind altogether as the PM completely encapsulates all of the behaviour for the view. This pattern is a very strong candidate for use in WPF applications and is also called Model-View-ViewModel.

There is a MSDN article about the Presentation Model and a section in the Composite Application Guidance for WPF (former Prism) about Separated Presentation Patterns

edited Dec 3 '17 at 23:01

community wiki
11 revs, 9 users 57%
Glenn Block

7 Can you please clarify this phrase? *This differs from MVP where actions route through the View to the Presenter. In MVC, every action in the View correlates with a call to a Controller along with an action.* To me, it sounds like the same thing, but I'm sure you're describing something different. – Panzercrisis Oct 19 '16 at 13:24

9 @Panzercrisis I'm not sure if this is what the author meant, but this is what I think they were trying to say. Like this answer - stackoverflow.com/a/2068/74556 mentions, in MVC, controller methods are based on behaviors -- in other words, you can map multiple views (but same behavior) to a single controller. In MVP, the presenter is coupled closer to the view, and usually results in a mapping that is closer to one-to-one, i.e. a view action maps to its corresponding presenter's method. You typically wouldn't map another view's actions to another presenter's (from another view) method. – Dustin Kendall Oct 28 '16 at 17:50

I blogged about this a while back, quoting on Todd Snyder's excellent post on the difference between the two:

> Here are the key differences between the patterns:
>
> **MVP Pattern**
>
> - View is more loosely coupled to the model. The presenter is responsible for binding the model to the view.
> - Easier to unit test because interaction with the view is through an interface
> - Usually view to presenter map one to one. Complex views may have multi presenters.
>
> **MVC Pattern**
>
> - Controller are based on behaviors and can be shared across views
> - Can be responsible for determining which view to display

It is the best explanation on the web I could find.

edited May 4 '15 at 3:28
Peter Mortensen

answered Aug 5 '08 at 10:21
Jon Limjap

 **12.3k** 18 81 107  **73.7k** 14 89 145

9 I don't understand how in the view can be coupled more or less closely to the model when in both cases the entire point is to completely decouple them. I'm not implying you said something wrong--just confused as to what you mean. – Bill K Oct 5 '11 at 0:25

10 @pst: with MVP it's really 1 View = 1 Presenter. With MVC, the Controller can govern multiple views. That's it, really. With the "tabs" model imagine each tab having its own Presenter as opposed to having one Controller for all tabs. – Jon Limjap Jun 29 '12 at 9:46

4 Originally there are two types of controllers: the one which you said to be shared across multiple views, and those who are views specific, mainly purposed for adapting the interface of the shared controller. – none Nov 11 '13 at 14:12

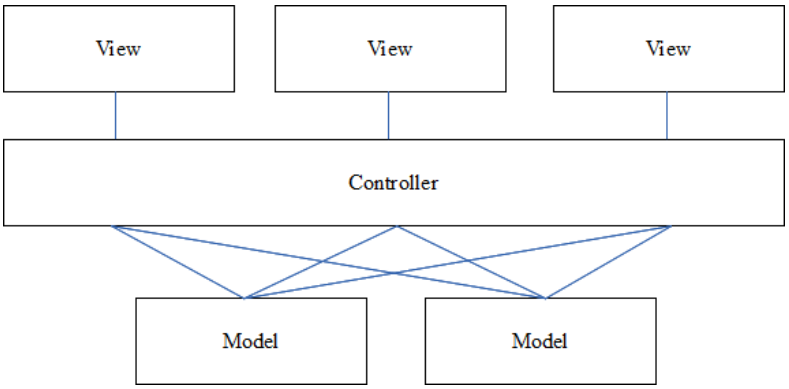
1 @JonLimjap What does it mean by one view anyway? In the context of iOS programming, is it one-screenful? Does this make iOS's controller more like MVP than MVC? (On the other hand you can also have multiple iOS controllers per screen) – huggie Mar 19 '14 at 7:55

1 Well Todd's diagramatic illustration of MVC completely contradicts the idea of decoupling the View and Model. If you look at the diagram, it says Model updates View (arrow from Model to View). In which universe is a system, where the Model directly interacts with the View, a decoupled one??? – Ash Jun 4 '17 at 2:01

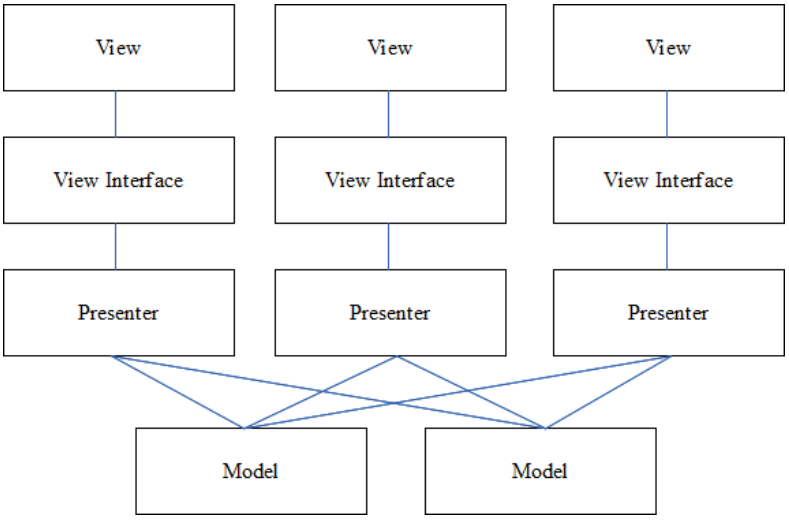
|

This is an oversimplification of the many variants of these design patterns, but this is how I like to think about the differences between the two.

MVC



MVP



answered Jul 6 '13 at 22:18

 Phyxx
9,697 8 46 71

8 This is a great depiction of the schematic, showing the abstraction and complete isolation of any GUI related (view stuff) from the API of the presenter. One minor point: A master presenter could be used where there is only one presenter, rather than one per view, but your diagram is the cleanest. IMO, the biggest difference between MVC/MVP is that MVP tries to keep the view totally void of anything other than display of the current 'view state' (view data), while also disallowing the view any knowledge of Model objects. Thus the interfaces, needing to be there, to inject that state. – user2080225 Oct 15 '13 at 3:30

3 Good picture. I use MVP quite a lot, so I'd like to make one point. In my experience, the Presenters need to talk

to one another quite often. Same is true for the Models (or Business objects). Because of these additional "blue lines" of communication that would be in your MVP pic, the Presenter-Model relationships can become quite entangled. Therefore, I tend to keep a one-to-one Presenter-Model relationship vs. a one-to-many. Yes, it requires some additional delegate methods on the Model, but it reduces many headaches if the API of the Model changes or needs refactoring. – splungebob Feb 28 '14 at 14:45

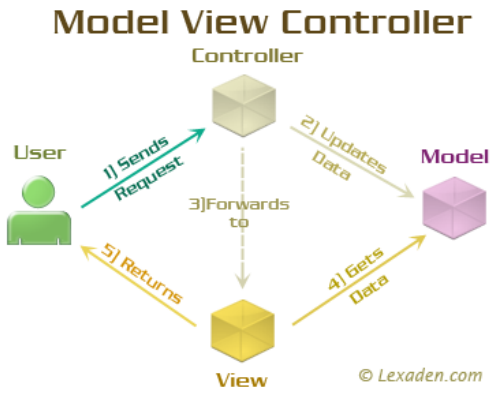
2 The MVC example is wrong; there's a strict 1:1 relationship between views and controllers. By definition, a controller interprets human gesture input to produce events for the model and view alike for a single control. More simply, MVC was intended for use with individual widgets only. One widget, one view, one control. – Samuel A. Falvo II Apr 5 '14 at 15:34

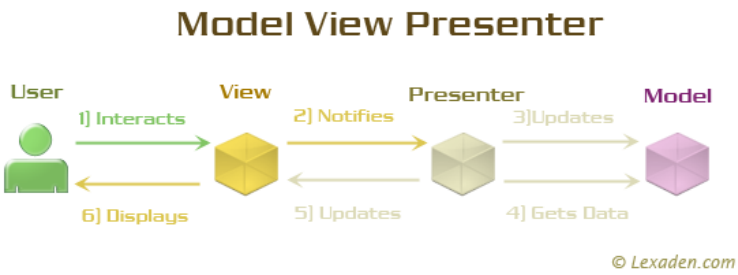
1 @SamuelA.FalvoII not always, there is a 1:Many between controllers and views in ASP.NET MVC: stackoverflow.com/questions/1673301/… – StuperUser Jan 7 '16 at 17:57

1 @StuperUser -- Not sure what I was thinking when I wrote that. You're right, of course, and looking back on what I wrote, I have to wonder if I had some other context in mind which I failed to articulate. Thanks for the correction. – Samuel A. Falvo II Jan 11 '16 at 23:40

|

Here are illustrations which represent communication flow





answered Sep 12 '14 at 20:47

Ashraf Bashir
6,190 9 39 74

25 I have a question regarding the MVC diagram. I don't get the part where the view goes out to fetch data .I would think the controller would forward to the view with the data needed. – Brian Rizo May 12 '15 at 21:07

35 If a user clicks a button, how is that not interacting with the view? I feel like in MVC, the user interacts with the view more than the controller – Jonathan Leaders Aug 12 '15 at 3:28

5 I know this is an old answer - but could anyone respond on @JonathanLeaders point? I'm coming from a winforms background unless you did some very unique coding, when you click the UI/View gets knowledge of that click before anything else. At least, as far as I know? – Rob P. Jan 4 '16 at 14:44

5 @RobP. I think these kinds of charts always tend to be either too complex, or too simple. Imo the flow of the MVP chart also holds true for a MVC application. There might be variations, depending on the languages features (data binding / observer), but in the end the idea is to decouple the view from the data/logic of the application. – Luca Fülbier Jan 17 '16 at 9:37

7 @JonathanLeaders People have *really* different things in mind when they say "MVC". Person who created this chart had probably classic Web MVC in mind, where the "user input" are HTTP requests, and "view returned to user" is a rendered HTML page. So any interaction between a user and a view are "not existent" from the perspective of an author of classical Web MVC app. – cubuspl42 Jun 12 '16 at 14:38

|

MVP is *not* necessarily a scenario where the View is in charge (see Taligent's MVP for example).
I find it unfortunate that people are still preaching this as a pattern (View in charge) as opposed to an

anti-pattern as it contradicts "It's just a view" (Pragmatic Programmer). "It's just a view" states that the final view shown to the user is a secondary concern of the application. Microsoft's MVP pattern renders re-use of Views much more difficult and *conveniently* excuses Microsoft's designer from encouraging bad practice.

To be perfectly frank, I think the underlying concerns of MVC hold true for any MVP implementation and the differences are almost entirely semantic. As long as you are following separation of concerns between the view (that displays the data), the controller (that initialises and controls user interaction) and the model (the underlying data and/or services)) then you are acheiving the benefits of MVC. If you are acheiving the benefits then who really cares whether your pattern is MVC, MVP or Supervising Controller? The only *real* pattern remains as MVC, the rest are just differing flavours of it.

Consider this highly exciting article that comprehensively lists a number of these differing implementations. You may note that they're all basically doing the same thing but slightly differently.

I personally think MVP has only been recently re-introduced as a catchy term to either reduce arguments between semantic bigots who argue whether something is truly MVC or not or to justify Microsofts Rapid Application Development tools. Neither of these reasons in my books justify its existence as a separate design pattern.

edited Feb 22 '13 at 16:35

answered Aug 25 '08 at 21:22

Quibblesome
20.1k 10 49 93

24 I've read several answers and blogs about the differences between MVC/MVP/MVVM/etc'. In effect, when you are down to business, it's all the same. It doesn't really matter whether you have an interface or not, and whether you are using a setter (or any other language feature). It appears that the difference between these patterns was born from the difference of various frameworks' implementations, rather than a matter of concept. – Michael Mar 7 '11 at 22:36

5 I wouldn't call MVP an *anti-pattern*, as later in the post "..the rest [including MVP] are just differing flavours of [MVC]..", which would imply that if MVP was an anti-pattern, so was MVC... it's just a flavor for a different framework's approach. (Now, some *specific* MVP implementations might be more or less desirable than some *specific* MVC implementations for different tasks...) – user166390 Jun 15 '12 at 19:31

@Quibblsome: “I personally think MVP has only been recently re-introduced as a catchy term to either reduce arguments between semantic bigots who argue whether something is truly MVC or not […] Neither of these reasons in my books justify its existence as a separate design pattern.” . It differs enough to make it distinct. In MVP, the view may be anything fulfilling a predefined interface (the View in MVP is a standalone component). In MVC, the Controller is made for a particular view (if relation's arities may make someone feel that's worth another term). – Hibou57 Feb 20 '13 at 15:50

5 @Hibou57, there is nothing to stop MVC from referencing the view as an interface or creating a generic controller for several different views. – Quibblesome Feb 22 '13 at 16:34

@quibblesome actually, there is. Controllers are, by definition, tightly bound to their corresponding views, for their job is to interpret human gestures (key presses, mouse updates, etc) into events for *individual controls* on a window. This is why you have a strict one controller, one view relationship. Controllers were *never* intended for form-wide use. For form-wide use, Application Model (aka Presentation Model) came into existence, which better suits that purpose. Since non-Smalltalk GUIs don't rely on MVC to implement controls, MVC makes relatively little sense in practice. – Samuel A. Falvo II Apr 5 '14 at 15:31

|

MVP: the view is in charge.

The view, in most cases, creates its presenter. The presenter will interact with the model and manipulate the view through an interface. The view will sometimes interact with the presenter, usually through some interface. This comes down to implementation; do you want the view to call methods on the presenter or do you want the view to have events the presenter listens to? It boils down to this: The view knows about the presenter. The view delegates to the presenter.

MVC: the controller is in charge.

The controller is created or accessed based on some event/request. The controller then creates the appropriate view and interacts with the model to further configure the view. It boils down to: the controller creates and manages the view; the view is slave to the controller. The view does not know about the controller.

edited May 4 '15 at 3:31

Peter Mortensen
12.3k 18 81 107

answered Aug 6 '08 at 22:51

Brian Leahy
14.8k 6 39 60

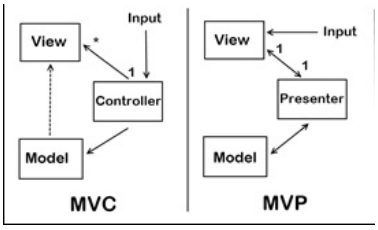
3 "View does not know about Controller." I think you mean that view has no contact directly with the model? – Lotus Notes Mar 25 '10 at 7:51

1 view should never know about the model in eiether one. – Brian Leahy Mar 29 '10 at 19:03

3 @Brian: “The View, in most cases, creates it's Presenter.” . I mostly seen the opposite, with the Presenter launching both the Model and the View. Well, the View may launch the Presenter too, but that point is not really

the most distinctive. What matters the most happens later during lifetime. – Hibou57 Feb 20 '13 at 15:55

You may want to edit your answer to explain further: Since the View does not know about the Controller, how are user actions, which are performed on the 'visual' elements the user sees on screen (i.e the View), communicated to the Controller... – Ash Jun 5 '17 at 5:21



MVC (Model View Controller)

The input is directed at the Controller first, not the view. That input might be coming from a user interacting with a page, but it could also be from simply entering a specific url into a browser. In either case, its a Controller that is interfaced with to kick off some functionality. There is a many-to-one relationship between the Controller and the View. That's because a single controller may select different views to be rendered based on the operation being executed. Note the one way arrow from Controller to View. This is because the View doesn't have any knowledge of or reference to the controller. The Controller does pass back the Model, so there is knowledge between the View and the expected Model being passed into it, but not the Controller serving it up.

MVP (Model View Presenter)

The input begins with the View, not the Presenter. There is a one-to-one mapping between the View and the associated Presenter. The View holds a reference to the Presenter. The Presenter is also reacting to events being triggered from the View, so its aware of the View its associated with. The Presenter updates the View based on the requested actions it performs on the Model, but the View is not Model aware.

For more Reference

answered Dec 20 '15 at 2:10

AVI
4,016 5 20 33

But in `MVP` pattern, when the application loads for the first time , isn't the presenter is responsible to load the first view? Like for example when we load the facebook applicaiton, isn't the presenter responsible to load the login page? – viper Nov 11 '16 at 5:18

A link from Model to View in MVC? You may want to edit your answer to explain how this makes it a 'decoupled' system, given this link. Hint: You may find it hard. Also, unless you think the reader will happily accept they've been computing wrong their whole life, you may want to elaborate on why actions go through Controller first in MVC despite the user interacting with the 'visual' elements on the screen (i.e the View), not some abstract layer that sits behind doing processing. – Ash Jun 5 '17 at 2:32

This should be the accepted answer. clear and concise – Nelson Ramirez Mar 14 at 2:00

Also worth remembering is that there are different types of MVPs as well. Fowler has broken the pattern into two - Passive View and Supervising Controller.

When using Passive View, your View typically implement a fine-grained interface with properties mapping more or less directly to the underlaying UI widget. For instance, you might have a ICustomerView with properties like Name and Address.

Your implementation might look something like this:

```
public class CustomerView : ICustomerView
{
    public string Name
    {
        get { return txtName.Text; }
        set { txtName.Text = value; }
    }
}
```

Your Presenter class will talk to the model and "map" it to the view. This approach is called the "Passive View". The benefit is that the view is easy to test, and it is easier to move between UI platforms (Web, Windows/XAML, etc.). The disadvantage is that you can't leverage things like databinding (which is *really* powerful in frameworks like WPF and Silverlight).

The second flavor of MVP is the Supervising Controller. In that case your View might have a property called Customer, which then again is databound to the UI widgets. You don't have to think about synchronizing and micro-manage the view, and the Supervising Controller can step in and help when needed, for instance with compled interaction logic.

The third "flavor" of MVP (or someone would perhaps call it a separate pattern) is the Presentation Model (or sometimes referred to Model-View-ViewModel). Compared to the MVP you "merge" the M and the P into one class. You have your customer object which your UI widgets is data bound to, but you also have additional UI-spesific fields like "IsButtonEnabled", or "IsReadOnly", etc.

I think the best resource I've found to UI architecture is the series of blog posts done by Jeremy Miller over at The Build Your Own CAB Series Table of Contents. He covered all the flavors of MVP and showed C# code to implement them.

I have also blogged about the Model-View-ViewModel pattern in the context of Silverlight over at *YouCard Re-visited: Implementing the ViewModel pattern*.

edited May 4 '15 at 3:34
Peter Mortensen
12.3k 18 81 107

answered Aug 8 '08 at 5:55
Jonas Follesø
4,570 7 31 52

- MVP = Model-View-Presenter
- MVC = Model-View-Controller

1. Both presentation patterns. They separate the dependencies between a Model (think Domain objects), your screen/web page (the View), and how your UI is supposed to behave (Presenter/Controller)
2. They are fairly similar in concept, folks initialize the Presenter/Controller differently depending on taste.
3. A great article on the differences is here. Most notable is that MVC pattern has the Model updating the View.

edited Apr 20 '13 at 9:17
Andrii Nemchenko
6,988 2 24 25

answered Aug 5 '08 at 10:22
Brett Veenstra
19.4k 16 58 77

Model updating the VIew. And this still is a decoupled system? – Ash Jun 5 '17 at 2:33

Both of these frameworks aim to seperate concerns - for instance, interaction with a data source (model), application logic (or turning this data into useful information) (Controller/Presenter) and display code (View). In some cases the model can also be used to turn a data source into a higher level abstraction as well. A good example of this is the MVC Storefront project.

There is a discussion here regarding the differences between MVC vs MVP.

The distinction made is that in an MVC application traditionally has the view and the controller interact with the model, but not with each other.

MVP designs have the Presenter access the model and interact with the view.

Having said that, ASP.NET MVC is by these definitions an MVP framework because the Controller accesses the Model to populate the View which is meant to have no logic (just displays the variables provided by the Controller).

To perhaps get an idea of the ASP.NET MVC distinction from MVP, check out this MIX presentation by Scott Hanselman.

edited Aug 5 '08 at 10:24

answered Aug 5 '08 at 10:20
Matt Mitchell
23.2k 29 98 170

7 MVC and MVP are patterns, not frameworks. If you honestly think, that topic was about .NET framework, then it is like hearing "the internet" and thinking it is about IE. – tereško Jun 18 '12 at 17:26

Pretty sure the question has evolved significantly from when it was first asked back in 2008. Additionally, looking back at my answer (and this was 4 years ago so I have not a lot more context than you) I'd say I start generally and then use .NET MVC as a concrete example. – Matt Mitchell Jun 19 '12 at 5:08

Both are patterns trying to separate presentation and business logic, decoupling business logic from UI aspects

Architecturally, MVP is Page Controller based approach where MVC is Front Controller based approach. That means that in MVP standard web form page life cycle is just enhanced by extracting the business logic from code behind. In other words, page is the one servicing http request. In other words, MVP IMHO is web form evolutionary type of enhancement. MVC on other hand changes completely the game because the request gets intercepted by controller class before page is loaded, the business logic is executed there and then at the end result of controller processing the data just dumped to the page ("view") In that sense, MVC looks (at least to me) a lot to Supervising Controller flavor of MVP enhanced with routing engine

Both of them enable TDD and have downsides and upsides.

Decision on how to choose one of them IMHO should be based on how much time one invested in ASP NET web form type of web development. If one would consider himself good in web forms, I would suggest MVP. If one would feel not so comfortable in things such as page life cycle etc MVC could be a way to go here.

Here's yet another blog post link giving a little bit more details on this topic

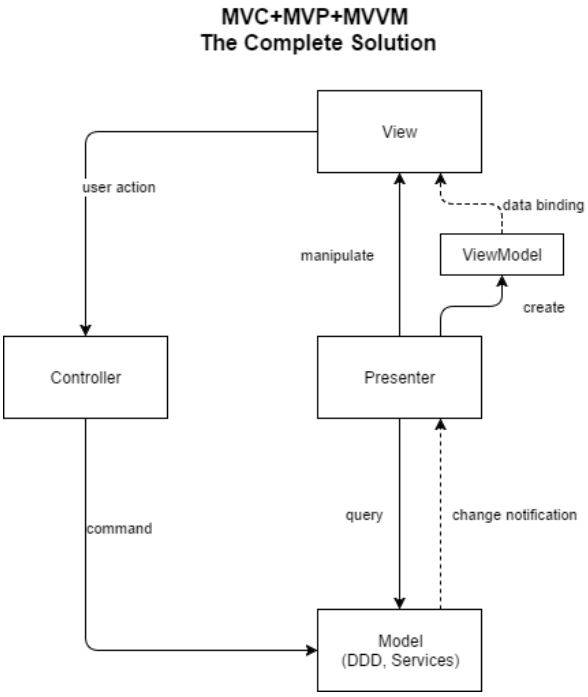
http://blog.vuscode.com/malovicn/archive/2007/12/18/model-view-presenter-mvp-vs-model-view-controller-mvc.aspx

edited Sep 22 '08 at 7:07

answered Sep 21 '08 at 12:32

Nikola Malovic
905 9 22

They each addresses different problems and can even be combined together to have something like below



There is also a complete comparison of MVC, MVP and MVVM here

answered Mar 13 '17 at 5:54

Xiaoguo Ge
894 7 14

Something I don't get is why data binding *has* to reduce testability. I mean, a view is effectively based off of what could be thought of as one or more database views, right? There might be connections between rows in different views. Alternatively, we can talk object-oriented instead of relational, but that actually doesn't change anything -- we still have one or more distinct data entities.

If we view programming as data structures + algorithms, then wouldn't it be best to have the data structures made explicit as possible, and then develop algorithms that each depend on as small a piece of data as possible, with minimal coupling between the algorithms?

I sense very Java-esque FactoryFactoryFactory thought patterns here -- we want to have multiple views, multiple models, multiple degrees of freedom all over the place. It's almost like that is the driving

force behind MVC and MVP and whatnot. Now let me ask this: how often is the cost you pay for this (and there most definitely *is* a cost) worth it?

I also see no discussion of how to efficiently manage state between HTTP requests. Haven't we learned from the functional folks (and the voluminous mistakes made by imperative spaghetti) that state is evil and should be minimized (and when used, should be well understood)?

I see a lot of usage of the terms *MVC* and *MVP* without much evidence that people think critically about them. Clearly, the problem is "them", me, or both...

answered Jan 28 '09 at 21:11
Luke Breuer

1 I've done MVC-ish work in very simple environments. It doesn't have to make things complex. It does give you very consistent guidelines for how to organize your code in a way that doesn't suck later. There's no reason to throw in excessive degrees of freedom - the M+V+C can be tightly coupled, and still good (because the M and V are independently testable). – Tom Jan 12 '11 at 14:41

I have used both MVP and MVC and although we as developers tend to focus on the technical differences of both patterns the point for MVP in IMHO is much more related to ease of adoption than anything else.

If I'm working in a team that already as a good background on web forms development style it's far easier to introduce MVP than MVC. I would say that MVP in this scenario is a quick win.

My experience tells me that moving a team from web forms to MVP and then from MVP to MVC is relatively easy; moving from web forms to MVC is more difficult.

I leave here a link to a series of articles a friend of mine has published about MVP and MVC.

http://www.qsoft.be/post/Building-the-MVP-StoreFront-Gutthrie-style.aspx

answered Jan 2 '09 at 10:35

Pedro Santos
779 1 13 21

In android there is version of mvc which is mvp: What is MVP?

The MVP pattern allows separate the presentation layer from the logic, so that everything about how the interface works is separated from how we represent it on screen. Ideally the MVP pattern would achieve that same logic might have completely different and interchangeable views.

First thing to clarify is that MVP is not an architectural pattern, it's only responsible for the presentation layer . In any case it is always better to use it for your architecture that not using it at all.

An example for mvp is https://github.com/antoniolg/androidmvp

What is MVC? MVC architecture is one of the oldest patterns available for achieving the separation of concerns. MVC consists of three layers, viz, Model, View, and Controller.

Classic MVC existed at a time when every control/gadget that existed in the screen was considered dumb and each control is paired with its own controller to manage the user interactions happening on them. So if 10 gadgets exist, then 10 controllers have to exist. In this scenario, every gadget is counted as a view. The advent of Windows GUI systems changed this picture. The Control-Controller relationship became obsolete. Controls gained the intelligence to respond to the actions initiated by the user. In the Windows world, view is a surface on which all the controls/gadgets exist, hence there is a need for only one controller. View can receive events and reach for controllers help to do further processing.

Sample code for mvc in android
http://androidexample.com/Use_MVC_Pattern_To_Create_Very_Basic_Shopping_Cart__-_Android_Example/index.php?view=article_discription&aid=116&aaid=138

Difference between both is available here http://www.codeproject.com/Articles/288928/Differences-between-MVC-and-MVP-for-Beginners

Now From my experience you must use MVP for android based project , because it enhanced version of MVC Model .

answered Jul 2 '16 at 7:15

Shubham Sharma
1,319 3 14 31

2 it has nothing to do with android and mvp is not a version of mvc – mikus Sep 13 '16 at 7:44

In MVP the view draws data from the presenter which draws and prepares/normalizes data from the model while in MVC the controller draws data from the model and set, by push in the view.

In MVP you can have a single view working with multiple types of presenters and a single presenter working with different multiple views.

MVP usually uses some sort of a binding framework, such as Microsoft WPF binding framework or various binding frameworks for HTML5 and Java.

In those frameworks, the UI/HTML5/XAML, is aware of what property of the presenter each UI element displays, so when you bind a view to a presenter, the view looks for the properties and knows how to draw data from them and how to set them when a value is changed in the UI by the user.

So, if for example, the model is a car, then the presenter is some sort of a car presenter, exposes the car properties (year, maker, seats, etc.) to the view. The view knows that the text field called 'car maker' needs to display the presenter Maker property.

You can then bind to the view many different types of presenter, all must have Maker property - it can be of a plane, train or what ever , the view doesn't care. The view draws data from the presenter - no matter which - as long as it implements an agreed interface.

This binding framework, if you strip it down, it's actually the controller :-)

And so, you can look on MVP as an evolution of MVC.

MVC is great, but the problem is that usually its controller per view. Controller A knows how to set fields of View A. If now, you want View A to display data of model B, you need Controller A to know model B, or you need Controller A to receive an object with an interface - which is like MVP only without the bindings, or you need to rewrite the UI set code in Controller B.

Conclusion - MVP and MVC are both decouple of UI patterns, but MVP usually uses a bindings framework which is MVC underneath. THUS MVP is at a higher architectural level than MVC and a wrapper pattern above of MVC.

edited May 4 '15 at 3:17

Peter Mortensen
12.3k 18 81 107

answered Jun 7 '13 at 21:16

James Roeiter
656 1 7 17

My humble short view: MVP is for large scales, and MVC for tiny scales. With MVC, I sometime feel the V and the C may be seen a two sides of a single indivisible component rather directly bound to M, and one inevitably falls to this when going down-to shorter scales, like UI controls and base widgets. At this level of granularity, MVP makes little sense. When one on the contrary go to larger scales, proper interface becomes more important, the same with unambiguous assignment of responsibilities, and here comes MVP.

On the other hand, this scale rule of a thumb, may weight very little when the platform characteristics favours some kind of relations between the components, like with the web, where it seems to be easier to implement MVC, more than MVP.

answered Feb 20 '13 at 16:55

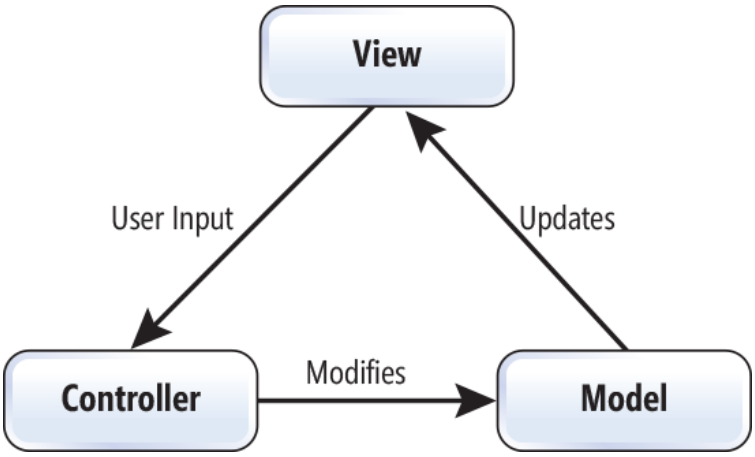
Hibou57
2,638 2 32 42

Model-View-Controller

MVC is a pattern for the architecture of a software application. It separate the application logic into three separate parts, promoting modularity and ease of collaboration and reuse. It also makes applications more flexible and welcoming to iterations.It separates an application into the following components:

- **Models** for handling data and business logic
- **Controllers** for handling the user interface and application
- **Views** for handling graphical user interface objects and presentation

To make this a little more clear, let's imagine a simple shopping list app. All we want is a list of the name, quantity and price of each item we need to buy this week. Below we'll describe how we could implement some of this functionality using MVC.

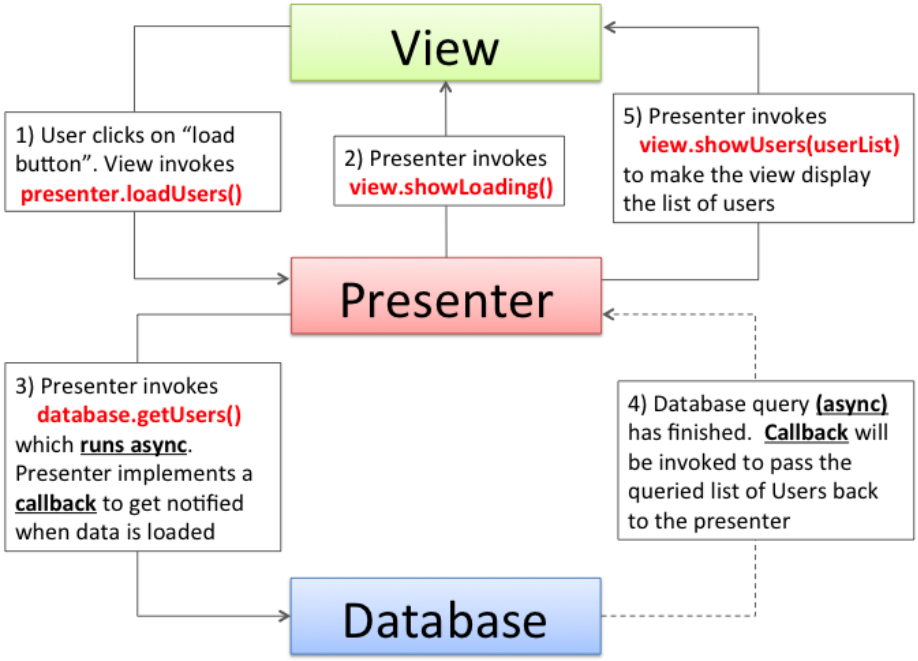


Model-View-Presenter

- The **model** is the data that will be displayed in the view (user interface).
- The **view** is an interface that displays data (the model) and routes user commands (events) to the Presenter to act upon that data. The view usually has a reference to its Presenter.
- The **Presenter** is the "middle-man" (played by the controller in MVC) and has references to both, view and model. **Please note that the word "Model"** is misleading. It should rather be **business logic that retrieves or manipulates a Model**. For instance: If you have a database storing User in a database table and your View wants to display a list of users, then the Presenter would have a reference to your database business logic (like a DAO) from where the Presenter will query a list of Users.

> If you want to see a sample with simple implementation please check this github post

A concrete workflow of querying and displaying a list of users from a database could work like this:



> What is the **difference** between **MVC** and **MVP** patterns?

MVC Pattern

- Controller are based on behaviors and can be shared across views
- Can be responsible for determining which view to display (Front Controller Pattern)

MVP Pattern

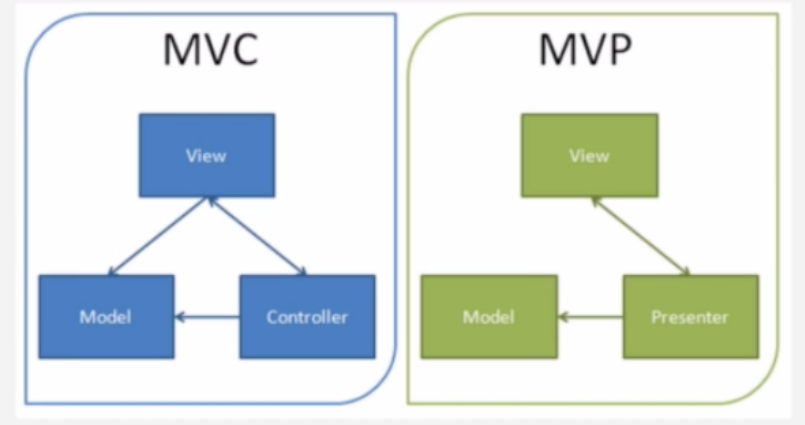
- View is more loosely coupled to the model. The presenter is responsible for binding the model to the view.
- Easier to unit test because interaction with the view is through an interface
- Usually view to presenter map one to one. Complex views may have multi presenters.

edited Feb 28 at 12:44

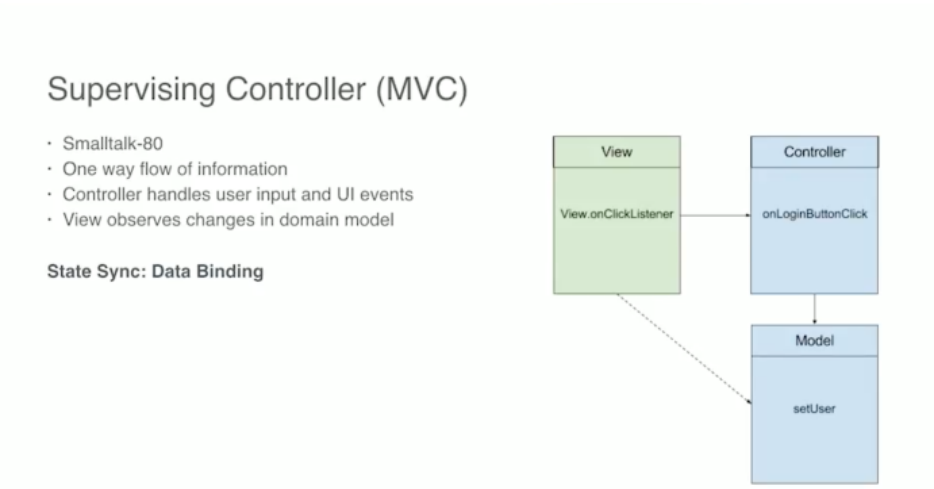
answered Nov 29 '17 at 10:14

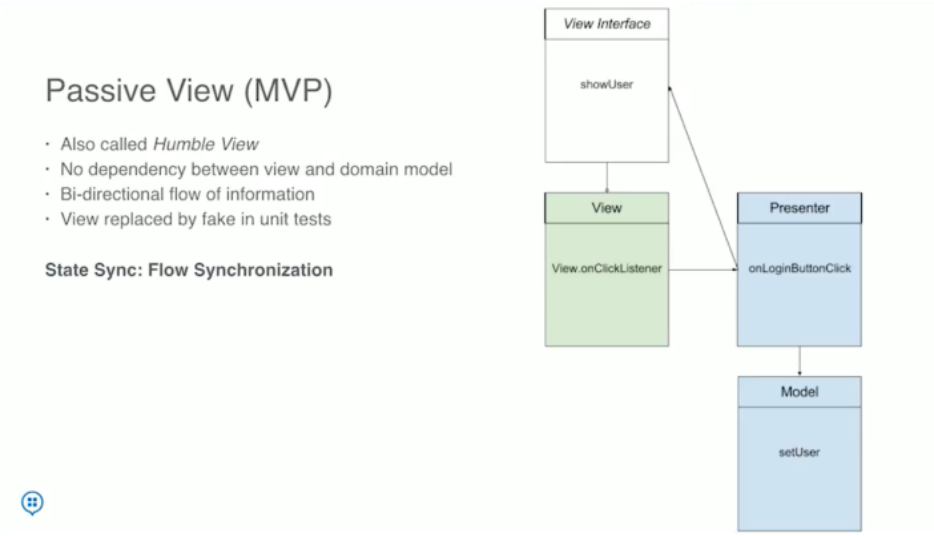
Rahul
234 3 16

There are many versions of MVC, this answer is about the original MVC in Smalltalk. In brief, it is



This talk droidcon NYC 2017 - Clean app design with Architecture Components clarifies it





edited Nov 14 '17 at 14:06

answered Sep 9 '15 at 2:34

onmyway133
20.8k 11 125 170

2 i've been implementing MVP all the time then... thinking it was MVC – R01010010 Sep 16 '15 at 12:15

5 In the MVC the Model is never called directly from the view – rodi Oct 29 '15 at 7:05

3 In the MVC the Model is never called directly from the View – rodi Oct 29 '15 at 7:05

4 This is an inaccurate answer. Do not be misled. as @rodi writes, there is no interaction between the View and Model. – Shawn Mehan Nov 18 '15 at 18:35

The MVC image is inaccurate or at best misleading, please do not pay any attention to this answer. – Jay1b Dec 7 '15 at 15:21

1 @Jay1b What MVC do you think is "correct"? This answer is about the original MVC. There's many other MVC (like in iOS) that was changed to adapt to the platform, say like `UIKit` – onmyway133 Nov 14 '17 at 14:08

There are many answers to the question, but I felt there is a need for some really simple answer clearly comparing the two. Here's the discussion I made up when a user searches for a movie name in an MVP and MVC app:

User: Click click …

View: Who's that? [**MVP**|**MVC**]

User: I just clicked on the search button …

View: Ok, hold on a sec … . [**MVP**|**MVC**]

(*View* calling the *Presenter*|*Controller* …) [**MVP**|**MVC**]

View: Hey *Presenter*|*Controller*, a User has just clicked on the search button, what shall I do? [**MVP**|**MVC**]

Presenter|*Controller*: Hey *View*, is there any search term on that page? [**MVP**|**MVC**]

View: Yes,… here it is … "piano" [**MVP**|**MVC**]

Presenter: Thanks *View*,… meanwhile I'm looking up the search term on the *Model*, please show him/her a progress bar [**MVP**|**MVC**]

(*Presenter*|*Controller* is calling the *Model* …) [**MVP**|**MVC**]

Presenter|*Controller*: Hey *Model*, Do you have any match for this search term?: "piano" [**MVP**|**MVC**]

Model: Hey *Presenter*|*Controller*, let me check … [**MVP**|**MVC**]

(*Model* is making a query to the movie database …) [**MVP**|**MVC**]

(After a while ...)

---------- This is where MVP and MVC start to diverge ----------

Model: I found a list for you, *Presenter*, here it is in JSON "[{"name":"Piano Teacher","year":2001}, {"name":"Piano","year":1993}]" [**MVP**]

Model: There is some result available, *Controller*. I have created a field variable in my instance and filled it with the result. It's name is "searchResultsList" [**MVC**]

(*Presenter*|*Controller* thanks *Model* and gets back to the *View*) [**MVP**|**MVC**]

Presenter: Thanks for waiting *View*, I found a list of matching results for you and arranged them in a presentable format: ["Piano Teacher 2001","Piano 1993"]. Please show it to the user in a vertical list. Also please hide the progress bar now [**MVP**]

Controller: Thanks for waiting *View*, I have asked *Model* about your search query. It says it has found a list of matching results and stored them in a variable named "searchResultsList" inside its instance. You can get it from there. Also please hide the progress bar now [**MVC**]

View: Thank you very much *Presenter* [**MVP**]

View: Thank you "Controller" [**MVC**] (Now the *View* is questioning itself: How should I present the results I get from the *Model* to the user? Should the production year of the movie come first or last...? Should it be in a vertical or horizontal list? ...)

In case you're interested, I have been writing a series of articles dealing with app architectural patterns (MVC, MVP, MVVP, clean architecture, ...) accompanied by a Github repo here. Even though the sample is written for android, the underlying principles can be applied to any medium.

answered Apr 6 at 13:51

Ali Nemati Hayati
853 10 19

MVP

MVP stands for Model - View- Presenter. This came to picture in early 2007 where Microsoft introduced Smart Client windows applications.

Presenter is acting as a supervisory role in MVP which binding View events and business logics from models.

View event binding will be implemented in Presenter from a view interface.

View is the initiator for user inputs and then delegates the events to Presenter and presenter handles event bindings and get data from models.

Pros: View is having only UI not any logics High level of testability

Cons: Bit complex and more work when implementing event bindings

MVC

MVC stands for Model-View-Controller. Controller is responsible for creating models and rendering views with binding models.

Controller is the initiator and it decides which view to render.

Pros: Emphasis on Single Responsibility Principle High level of testability

Cons: Sometimes too much workload for Controllers, if try to render multiple views in same controller.

answered Jan 12 '16 at 4:50

marvelTracker
1,724 16 38

Actually, MVP pattern is the successor of older Model View Controller (MVC) architecture. They created this, because there were problems with redesign, code modification, testing and better UI. This patterns divides the application code in three layers.

1. The Model – accommodating models (POJO, JavaBeans, etc.), data from various sources (database, server results, cache, Android file system, etc.).
2. The View is responsible for visual display of applications, along with data input by users. This part should NOT under any circumstances process the data. Its function is only to detect the input (e.g. touch or swipe) and visualization.
3. Presenter, which is responsible for passing of data between two layers, and thus the analysis and interaction. Such division allows substitution of code parts, proper testing and connects layers as interfaces.

answered Feb 23 at 12:10

Anzeem S N
11 6

There is this nice video from Uncle Bob where he briefly explains MVC & MVP at the end.

What I think is MVP is an improved version of MVC where you basically separate the concern of how you're gonna show the data you have. Presenter includes kinda the business logic of your UI and it talks through an interface which makes it easier to test your UI business logic across different views. MVC still talks through interfaces (boundaries) to glue layers but it has no such restriction to impose UI presentation logic. Other than that, I don't really see any more differences.

answered Jan 25 at 21:24

zgulser
640 1 9 18

A lot of people does not know exactly what is the difference between the Controller and Presenter in MVC and MVP respectively.

its a simple equation where

MVC View = View and Presenter of MVP

MVP Model = Controller and Model of MVC

more info refer to this http://includeblogh.blogspot.com.eg/2016/07/the-difference-and-relationship-between.html

answered Jul 31 '17 at 10:13

AhmedNTS


101 1 7

The simplest answer is how the view interacts with the model. In MVP the model is bound to the presenter, which is responsible for updating the view. In MVC the model updates the view directly.

edited Mar 16 at 10:21

answered Nov 16 '17 at 17:32


Clive Jefferies
672 7 23