

# What are MVP and MVC and what is the difference?

Asked 16 years, 4 months ago    Modified 3 months ago

Viewed 577k times



2382



**Want to improve this post?** Provide detailed answers to this question, including citations and an explanation of why your answer is correct. Answers without enough detail may be edited or deleted.



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?

user-interface

model-view-controller

design-patterns

terminology

mvp

Share

edited Aug 29, 2020 at 1:23

Improve this question

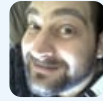
Follow



Wicket

37.9k ● 9 ● 77 ● 188

asked Aug 5, 2008 at 10:06



Mike Minutillo

54.8k ● 14 ● 48 ● 41

---

6 [mvc.givan.se/#mvp](https://mvc.givan.se/#mvp) – givanse Dec 11, 2016 at 16:51

---

3 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, 2017 at 1:19

---

1 MVP adds an extra layer of indirection by splitting the View-Controller into a View and a Presenter... – [the\\_prole](#) Jan 26, 2018 at 2:33

---

4 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](https://digigene.com/category/android/android-architecture) – [Ali Nem](#) Mar 19, 2018 at 11:49

---

1 They are called **architectural patterns** not **design patterns**. If you want to know the difference, check [this](#) – [Hasan El-Hefnawy](#) Jun 2, 2019 at 22:17

---

24 Answers

Sorted by:

Highest score (default)





2134



## Model-View-Presenter

In **MVP**, the Presenter contains the UI business logic for the View. All invocations from the View delegate directly to the 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 WebForms. 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. A 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

that 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 data binding the amount of code is reduced.
- Con: there's a 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 behavior 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](#)

Share Improve this answer

edited May 19, 2021 at 14:23

Follow

community wiki

14 revs, 12 users 51%

Glenn Block

---

42 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, 2016 at 13:24

---

20 @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](https://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, 2016 at 17:50

---

- 4 **Note that** `MVC` is often used by web-frameworks like `Laravel`, where the received URL requests (maybe made by users) are handled by the `Controller` and the HTML generated by the `View` is sent to the client -- So, the `View` is a part of the *backend* and the user can never access it directly, and if you experience anywhere the opposite then consider that as an MVC-extension (or even violation). @PanzerCrisis, This differs from `MVP` (like that used in `Android OS`) where actions route through the `View` to the `Presenter` and user have direct access to the `View`. – [Top-Master](#) Sep 29, 2019 at 11:50



- 7 What the author describes when speaking about MVC isn't the original Smalltalk MVC (whose flow is triangular) but the "Web MVC" where the controller renders a view using a model and returns it to the user. I believe this is worth noting because this creates a lot of confusion. – [raiks](#) Jul 15, 2020 at 8:47



Is there any **fundamental** difference between Presentation Model (MVVM) and MVP Passive View (MVP-PV)? 1. MVP-PV: "View exposes setter properties that the Presenter uses to set the data", MVVM: "The View then subscribes to events coming from the Presentation Model and updates itself accordingly" – the same flow of information, difference in implementation details. 2. "The Presentation Model is a Model crafted specifically for the View" – just like in MVP Passive View (the view is dumb, so the Presentation Model must be crafted to it) – [kxmh42](#) Oct 13, 2023 at 15:24

---

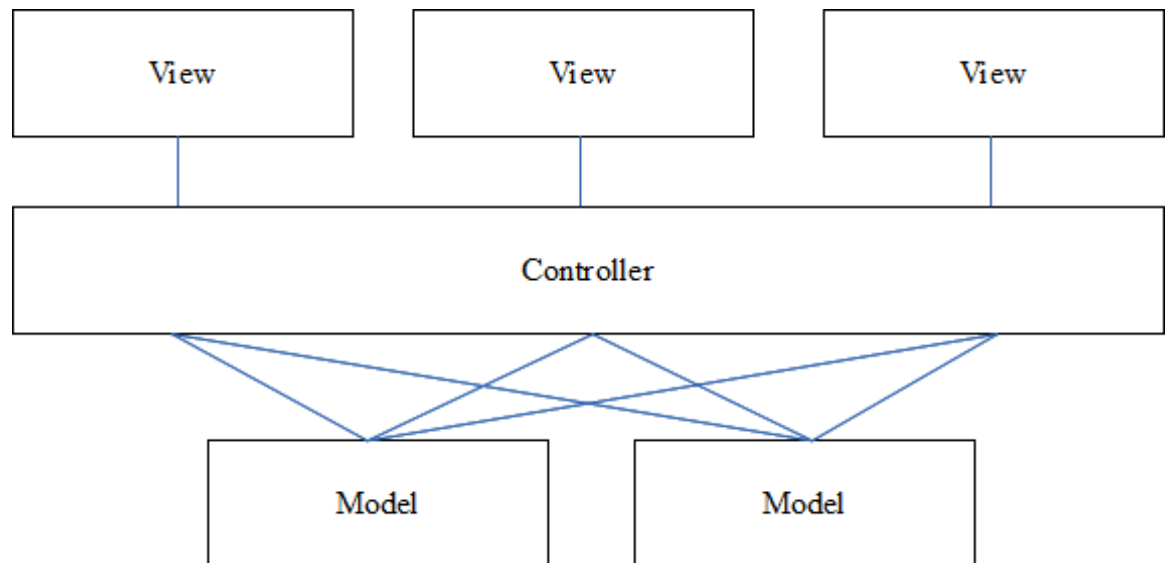


522

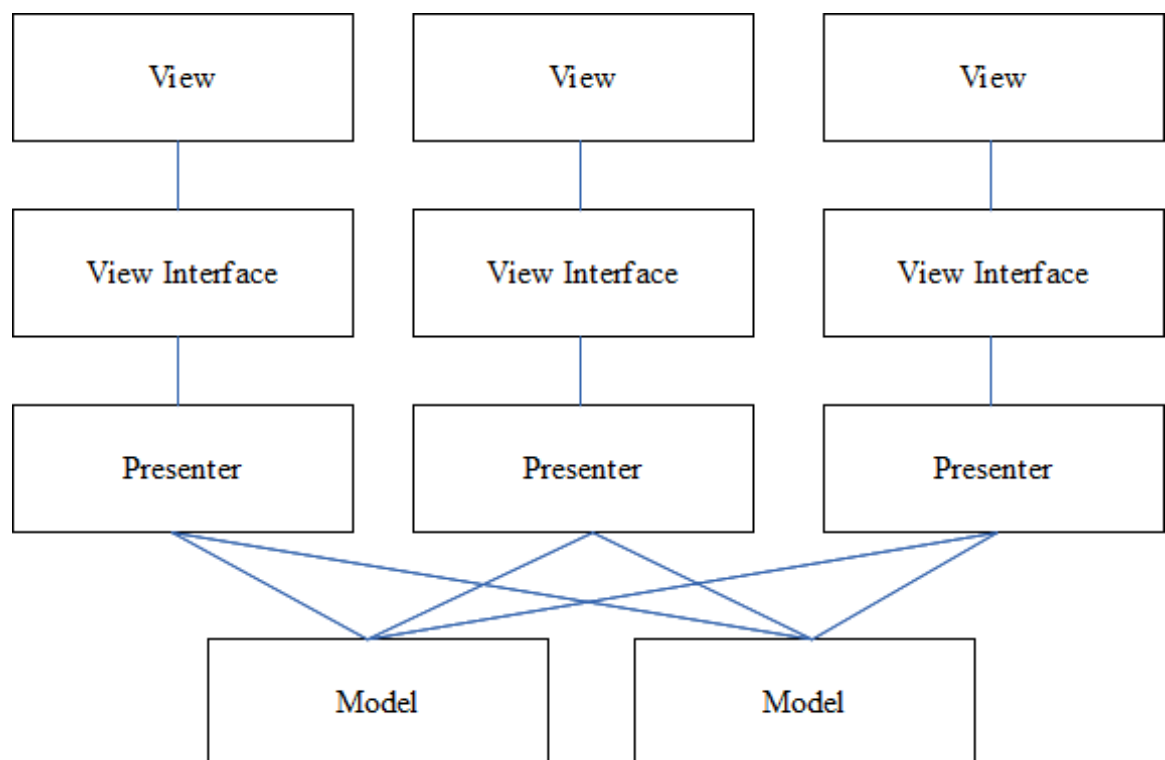


This is an oversimplification of the many variants of these design patterns, but this is how I like to think about the differences between the two.

## MVC



## MVP







---

12 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, 2013 at 3:30

---

4 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, 2014 at 14:45

---

4 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, 2014 at 15:34

---

3 @SamuelA.FalvoII not always, there is a 1:Many between controllers and views in ASP.NET MVC:

- 6 @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, 2016 at 23:40



449

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.

Share Improve this answer

Follow

edited May 4, 2015 at 3:28



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Aug 5, 2008 at 10:21



Jon Limjap

95.3k ● 15 ● 103 ● 153

---

19 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, 2011 at 0:25

---

20 @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, 2012 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. – [Acsor](#) Nov 11, 2013 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, 2014 at 7:55

---

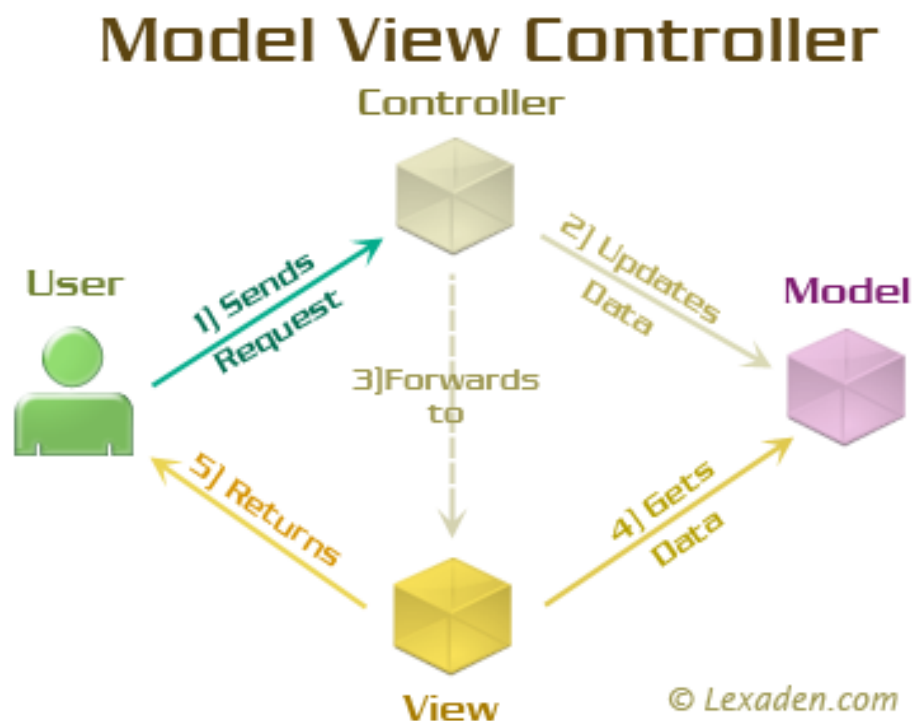
- 2 Well Todd's diagrammatic illustration of MVC completely contradicts the idea of decoupling the View and Model. If you look at the diagram, it says Model updates View (arrow from Model to View). In which universe is a system, where the Model directly interacts with the View, a decoupled one???
- [Ash](#) Jun 4, 2017 at 2:01



286



Here are illustrations which represent communication flow



### Model View Presenter



© Lexaden.com

**Note** that the `MVC` Chart is from the backend's perspective, **meaning**:

- No matter if `Angular` or similar frontend-framework generates the HTML, or if classic backend-generated-HTML is used,
- The `View` lives in the backend, even if the `View` generates `JSON` for `Angular`'s needs instead of `HTML`.
- Hence, no matter what button the user clicks, the user is clicking on the rendered-output, never the actual `View` ;-)

Share Improve this answer

Follow

edited Aug 30 at 3:53



Top-Master

8,679 ● 5 ● 47 ● 85


answered Sep 12, 2014 at 20:47



Ashraf Bashir

9,804 ● 15 ● 60 ● 83

---

52 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, 2015 at 21:07 

---

72 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](#) Aug 12, 2015 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, 2016 at 14:44

---

7 @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, 2016 at 9:37

---

26 @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, 2016 at 14:38

---



189



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 achieving the benefits of MVC. If you are achieving 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.

Share Improve this answer

edited Apr 2, 2020 at 17:05

Follow

answered Aug 25, 2008 at 21:22




Quibblesome


25.4k ● 10 ● 62 ● 104

---

**33** 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, 2011 at 22:36 

---

- 6 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, 2012 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, 2013 at 15:50 

---

- 7 @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, 2013 at 16:34
- 

- 2 Samuel please clarify what you're talking about. Unless you're telling me the history of the team that "invented" MVC then I'm incredibly dubious about your text. If you're just talking about WinForm then there are other ways of doing things and I've created WinForm projects where control bindings are managed by the controller, not "individual controls". – [Quibblesome](#) Apr 7, 2014 at 12:44
-





123



## 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.

Share Improve this answer

Follow

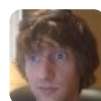
edited Jun 20, 2020 at 9:12



Community Bot

1 • 1

answered Aug 6, 2008 at 22:51



Brian Leahy

35.4k • 12 • 46 • 60

- 
- 3 "View does not know about Controller." I think you mean that view has no contact directly with the model? – [Lotus Notes](#)

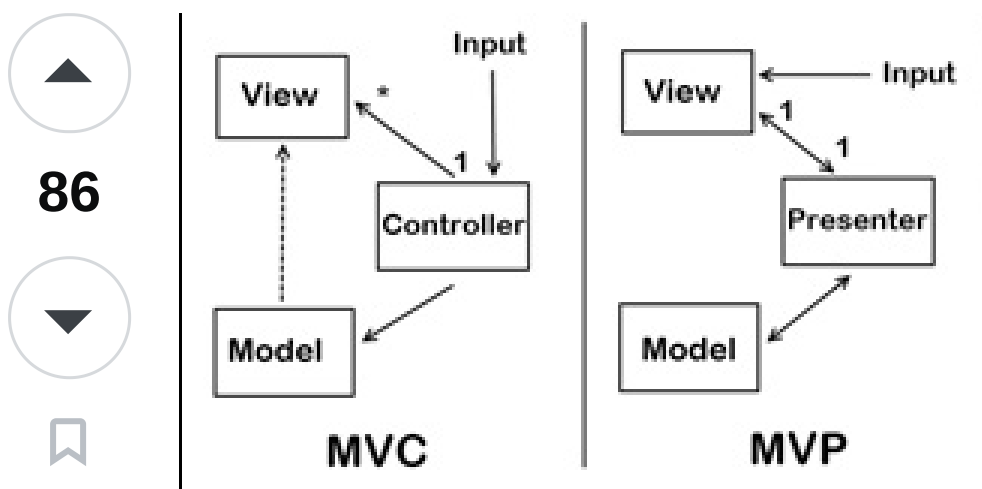
2 view should never know about the model in eiether one.

– [Brian Leahy](#) Mar 29, 2010 at 19:03

5 @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, 2013 at 15:55

2 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, 2017 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](#)

Share Improve this answer

answered Dec 20, 2015 at 2:10

Follow



AVI

5,693 ● 5 ● 31 ● 40

---

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? – [vipser](#) Nov 11, 2016 at 5:18

---

2 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, 2017 at 2:32

---

5 This Is clearly wrong... in MVC, model never talks directly with view and vice versa. They don't even know other one exists. The controller is the glue that holds them together – [MegaManX](#) Oct 25, 2018 at 12:39

---

1 I agree with Ash and MegaManX. In the MVC diagram, the arrow should be from the View pointing to the Model (or ViewModel, or DTO), not from Model to the View; because the Model does not know about the View, but the view might know about the Model. – [jflaga](#) Apr 10, 2019 at 6:45

---

1 Actually, I think based on the original SmallTalk triangular MVC, the Model-View link is correct: [commons.wikimedia.org/wiki/File:MVC-Process.svg#/media/...](https://commons.wikimedia.org/wiki/File:MVC-Process.svg#/media/...) . The problem I see is the input to the Controller and its link to the View. Normally the user interacts with the view, so the View should be linked towards the Controller. – [Mahm00d](#) Oct 17, 2021 at 8:46

---



**83**



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|Controller*: 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.

Share Improve this answer

Follow

edited May 9, 2021 at 12:17



Ray Jasson

451 ● 2 ● 10 ● 28

answered Apr 6, 2018 at 13:51



Ali Nem

5,530 ● 1 ● 44 ● 42

---

1 Basically what you're trying to say is that the controller micromanages the view logic? So it makes the view dumber by presenting what happens and how on views? – [Radu](#) Aug 28, 2018 at 14:37

---

1 @Radu, No, it does not micromanage, that is what the presenter does by making the view passive or dumb – [Ali Nem](#) Aug 28, 2018 at 22:48

---

9 In a proper MVC, the view invokes functionality on the controller, and listens to data changes in the model. The view does not get data from the controller, and the controller should NOT tell the view to display, for instance, a loading indicator. A proper MVC allows you to replace the view part, with one that is fundamentally different. The view part holds view logic, that includes a loading indicator. The view invokes instructions (in controller), controller modifies data in the model, and the model notifies its listeners of changes to its



46



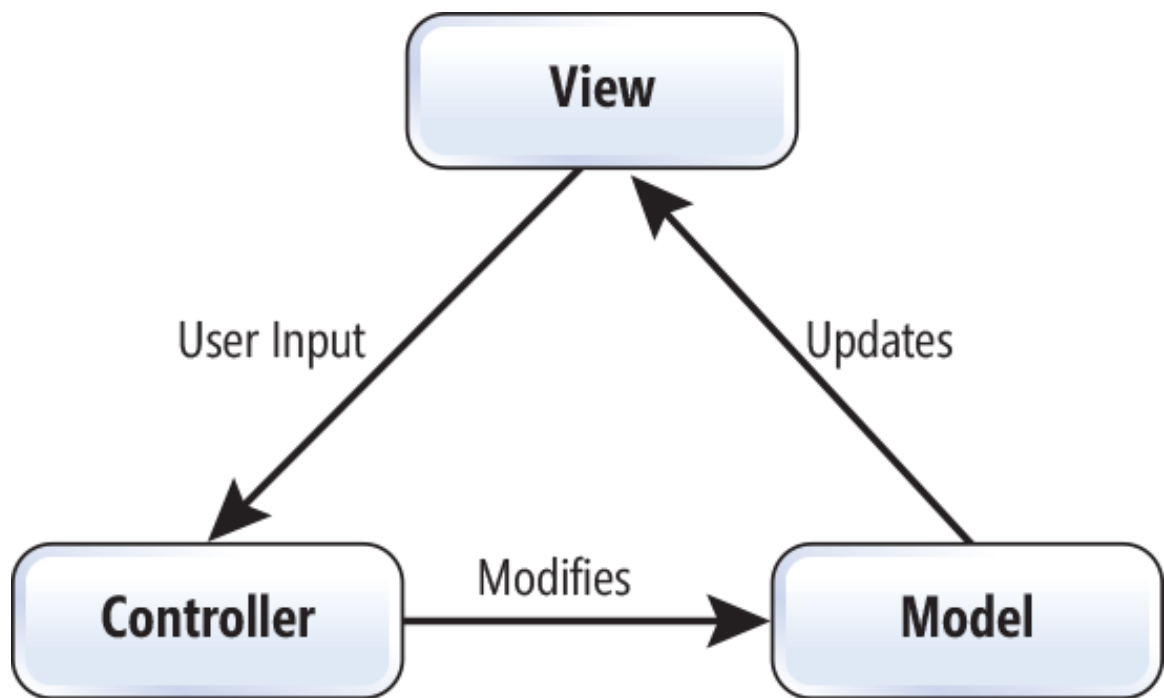
## Model-View-Controller

**MVC** is a pattern for the architecture of a software application. It separates 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.



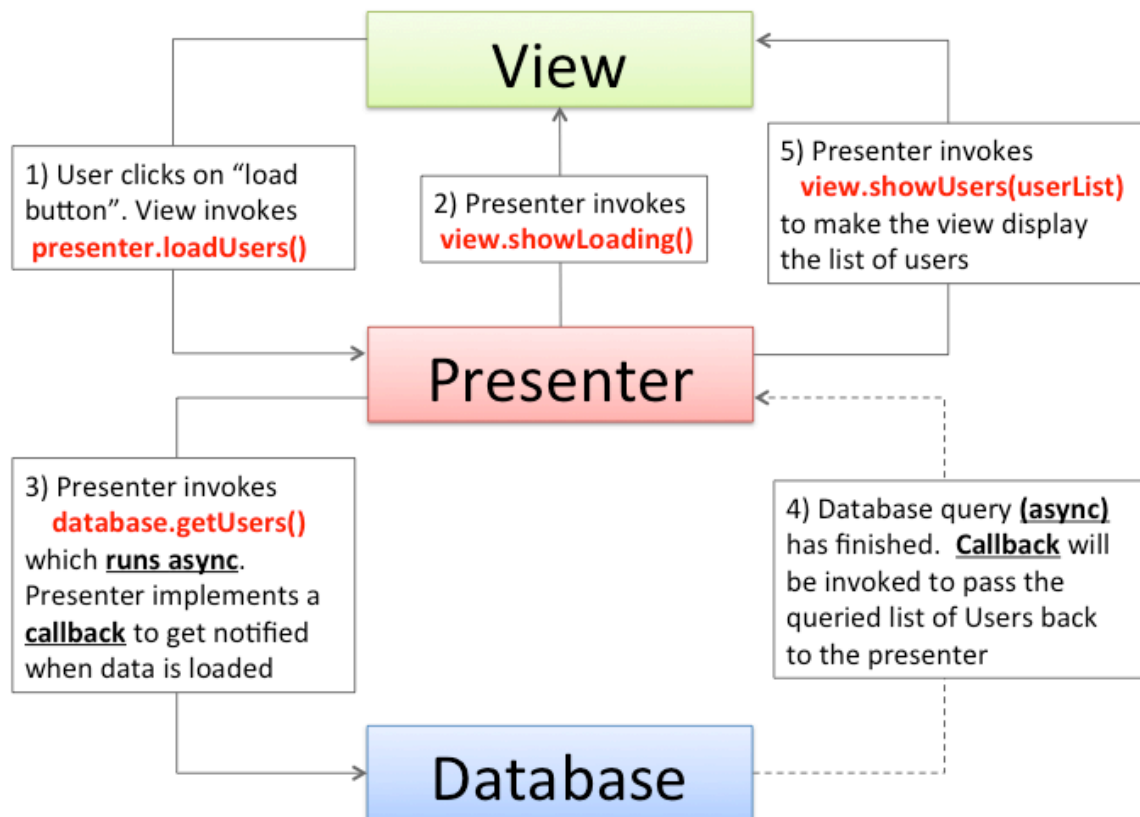


## Model-View-Presenter

- 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.

Share Improve this answer

edited Jun 13, 2019 at 7:38

Follow

answered Nov 29, 2017 at 10:14



Rahul

3,349 ● 2 ● 34 ● 44



38



- 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.

Share Improve this answer

edited Apr 20, 2013 at 9:17

Follow



user355491

answered Aug 5, 2008 at 10:22



Brett Veenstra

48.4k ● 18 ● 71 ● 86

---

3 Model updating the View. And this still is a decoupled system? – [Ash](#) Jun 5, 2017 at 2:33

---



37

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 underlying 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 complex 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-specific 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](#).

Share Improve this answer

edited May 4, 2015 at 3:34

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Aug 8, 2008 at 5:55



Jonas Follesø

6,531 ● 7 ● 42 ● 54



21



Both of these frameworks aim to separate 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.

Share Improve this answer

edited Aug 5, 2008 at 10:24

Follow


answered Aug 5, 2008 at 10:20



Matt Mitchell

41.8k ● 35 ● 121 ● 185

---

9 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, 2012 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, 2012 at 5:08

---



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

14

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>



Share Improve this answer

Follow

answered Sep 21, 2008 at 12:32



Nikola Malovic

1,250 ● 1 ● 13 ● 24



10



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>

Share Improve this answer

Follow

answered Jan 2, 2009 at 10:35



Pedro Santos

1,004 ● 2 ● 15 ● 23



9



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.

Share Improve this answer

Follow

edited May 4, 2015 at 3:17



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jun 7, 2013 at 21:16



James Roeiter

861 ● 1 ● 10 ● 23



## MVC (Model-View-Controller)

9

In MVC, the Controller is the one in charge! The Controller is triggered or accessed based on some



events/requests then, manages the Views.



Views in MVC are virtually stateless, the Controller is responsible for choosing which View to show.



**E.g.:** When the user clicks on the “Show MyProfile” button, the Controller is triggered. It communicates with the Model to get the appropriate data. Then, it shows a new View that resembles the profile page. The Controller may take the data from the Model and feed it directly to the View -as proposed in the above diagram- or let the View fetch the data from the Model itself.

## MVP (Model-View-Presenter)

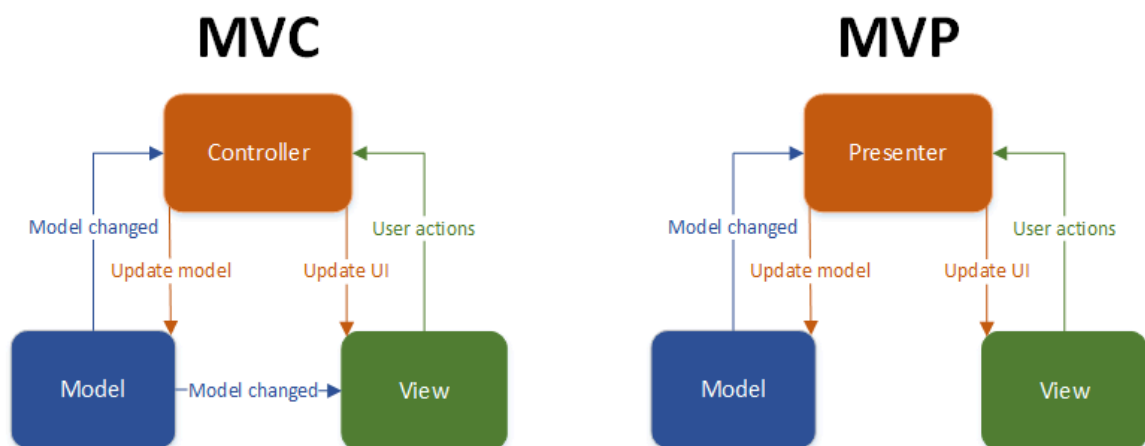
In MVP, the View is the one in charge! each View calls its Presenter or has some events that the Presenter listens to.

Views in MVP don't implement any logic, the Presenter is responsible for implementing all the logic and communicates with the View using some sort of interface.

**E.g.:** When the user clicks the “Save” button, the event handler in the View delegates to the Presenter's “OnSave” method. The Presenter will do the required logic and any needed communication with the Model then, calls back the View through its interface so that the View can display that the save has been completed.

# MVC vs. MVP

- MVC doesn't put the View in charge, Views act as slaves that the Controller can manage and direct.
- In MVC, Views are stateless contrary to Views in MVP where they are stateful and can change over time.
- In MVP, Views have no logic and we should keep them dumb as possible. On the other hand, Views in MVC may have some sort of logic.
- In MVP, the Presenter is decoupled from the View and talks to it through an interface. This allows mocking the View in unit tests.
- In MVP, Views are completely isolated from the Model. However, in MVC, Views can communicate with the Model to keep it up with the most up-to-date data.



Share Improve this answer

edited Jan 27, 2022 at 6:54

Follow

answered Oct 26, 2021 at 23:59



Mostafa Wael

3,753 ● 1 ● 25 ● 32



7



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.

Share Improve this answer

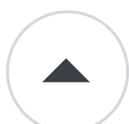
Follow

answered Feb 20, 2013 at 16:55



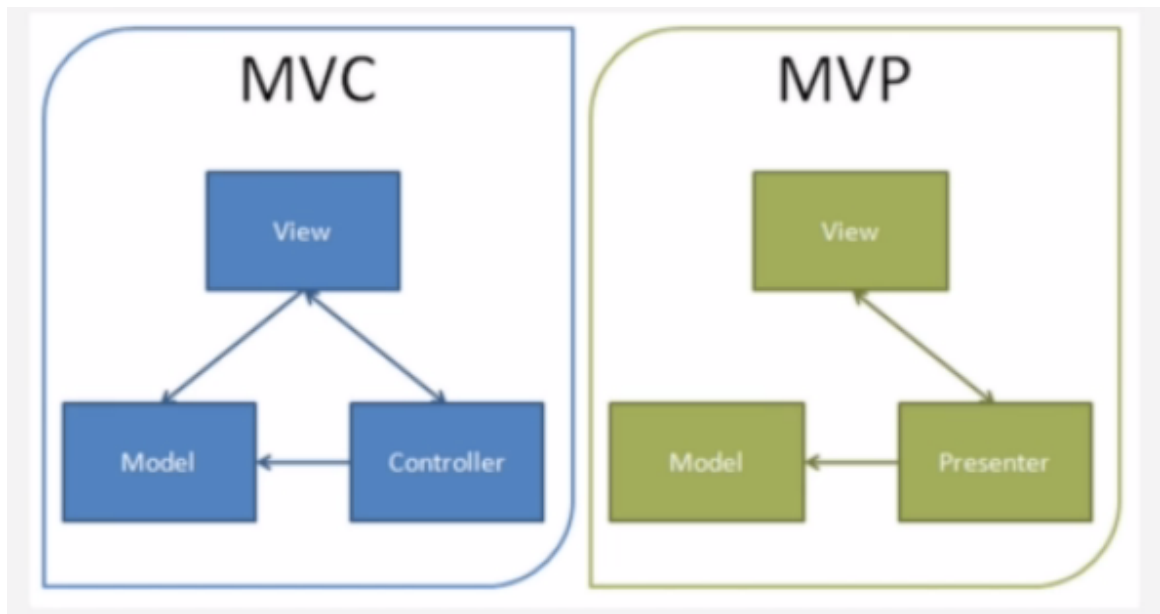
Hibou57

7,140 ● 8 ● 54 ● 58



7

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

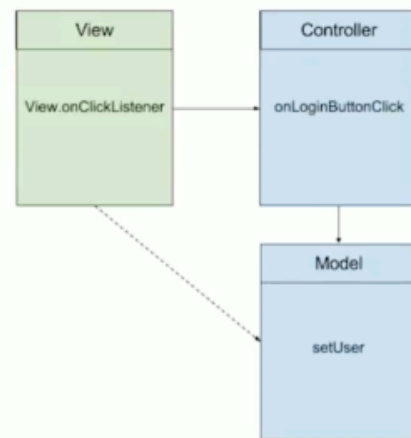


This talk [droidcon NYC 2017 - Clean app design with Architecture Components](#) clarifies it

## Supervising Controller (MVC)

- Smalltalk-80
- One way flow of information
- Controller handles user input and UI events
- View observes changes in domain model

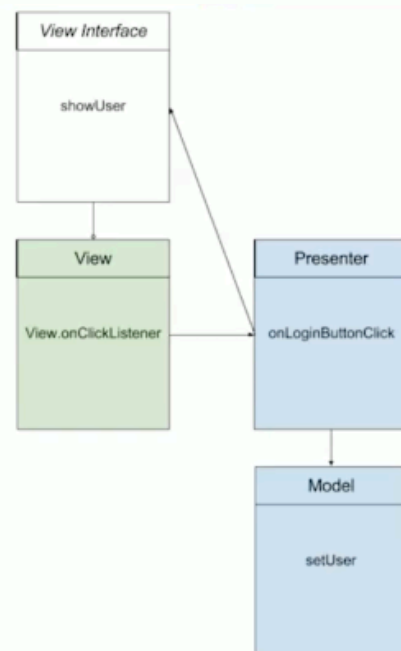
**State Sync: Data Binding**



## Passive View (MVP)

- Also called *Humble View*
- No dependency between view and domain model
- Bi-directional flow of information
- View replaced by fake in unit tests

**State Sync: Flow Synchronization**



Share Improve this answer

edited Nov 14, 2017 at 14:06

Follow

answered Sep 9, 2015 at 2:34



[onmyway133](#)

47.9k ● 32 ● 266 ● 271

6 In the MVC the Model is never called directly from the view

– [M4rk](#) Oct 29, 2015 at 7:05



- 5 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, 2015 at 18:35

---

The MVC image is inaccurate or at best misleading, please do not pay any attention to this answer. – [Jay](#) Dec 7, 2015 at 15:21

- 
- 4 @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, 2017 at 14:08 ✎

---

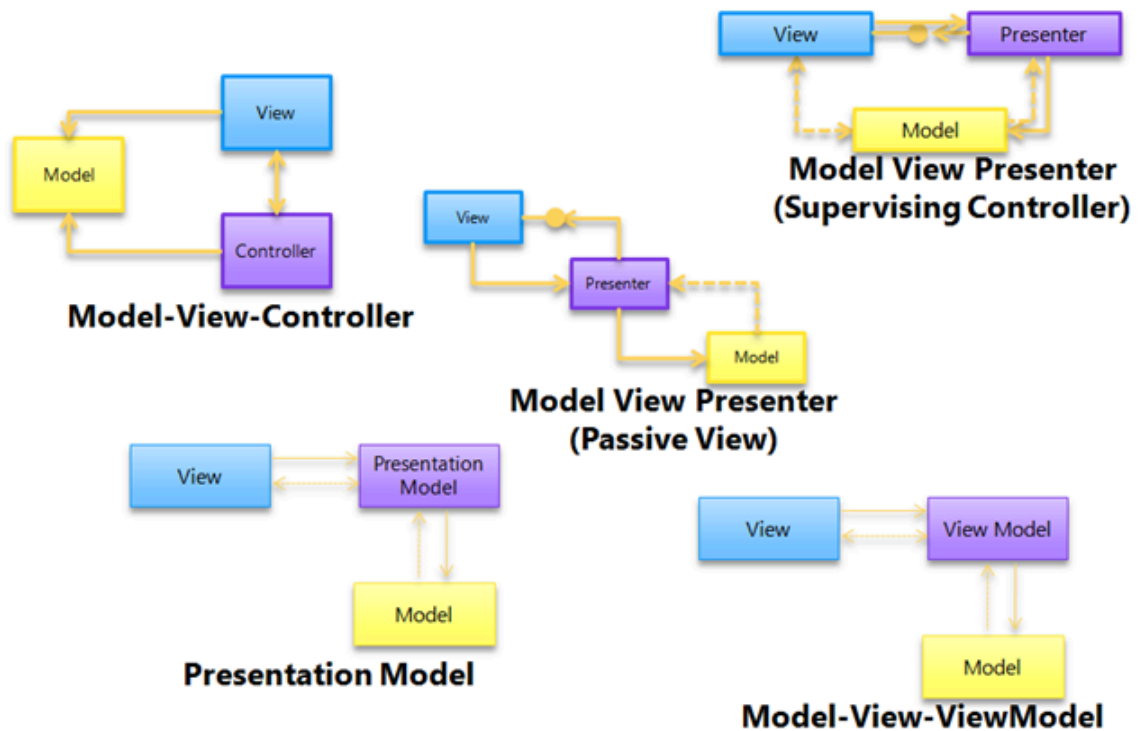
What do the arrows mean? – [problemofficer - n.f. Monica](#)  
Nov 10, 2018 at 16:04



7



I think this image by Erwin Vandervalk (and the accompanying [article](#)) is the best explanation of MVC, MVP, and MVVM, their similarities, and their differences. The [article](#) does not show up in search engine results for queries on "MVC, MVP, and MVVM" because the title of the article does not contain the words "MVC" and "MVP"; but it is the best explanation, I think.



(The [article](#) also matches what Uncle Bob Martin said in his one of his talks: that MVC was originally designed for the small UI components, not for the architecture of the system)

Share Improve this answer

answered May 10, 2019 at 1:43

Follow



jflaga

4,749 ● 2 ● 27 ● 21



5



The simplest answer is how the view interacts with the model. In MVP the view is updated by the presenter, which acts as an intermediary between the view and the model. The presenter takes the input from the view, which retrieves the data from the model and then performs any business logic required and then updates the view. In MVC the model updates the view directly rather than going back through the controller.

answered Nov 16, 2017 at 17:32

**Clive Jefferies**

1,158 ● 14 ● 26

---

I have downvoted, because afaik the model does not know anything about the view in MVC and there is not able to update it directly as you write. – [problemofficer - n.f. Monica](#) Nov 10, 2018 at 15:58 ✎

---

2 Look at MVC on Wikipedia, that is exactly how it works. – [Clive Jefferies](#) Nov 11, 2018 at 22:43

---

2 Whether readers like it or not, plenty sources that can be found by googling state that in MVC the view subscribes to updates on the model. *and* in some cases might even *be* the controller and hence *invoke* such updates. If you don't like that, then go complain on those articles, or cite which 'bible' you think is the sole legitimate source, instead of downvoting answers that just relay the other info available out there! – [underscore\\_d](#) Dec 8, 2018 at 10:21

---

1 The wording could definitely be improved, but it's true that the view subscribes to changes in the model in MVC. The model does not need to know the View in MVC. – [devoured elysium](#) Jan 6, 2019 at 12:50 ✎

---



There is [this](#) nice video from Uncle Bob where he briefly explains **MVC** & **MVP** at the end.

5



IMO, MVP is an improved version of MVC where you basically separate the concern of what you're gonna show (the data) from how you're gonna show (the view).



The presenter includes kinda the business logic of your UI, implicitly imposes what data should be presented and gives you a list of dumb view models. And when the time comes to show the data, you simply plug your view (probably includes the same id's) into your adapter and set the relevant view fields using those view models with a minimum amount of code being introduced (just using setters). Its main benefit is you can test your UI business logic against many/various views like showing items in a horizontal list or vertical list.

In MVC, we talk through interfaces (boundaries) to glue different layers. A controller is a plug-in to our architecture but it has no such a restriction to impose what to show. In that sense, MVP is kind of an MVC with a concept of views being pluggable to the controller over adapters.

I hope this helps better.

Share Improve this answer

Follow

edited Mar 31, 2020 at 20:58



Rahul


3,349 ● 2 ● 34 ● 44

answered Jan 25, 2018 at 21:24



stdout

2,601 ● 2 ● 33 ● 44

- 
- 3 Important point from Uncle Bob: When originally invented by Trygve Reenskaug, MVC was meant for *each widget* not the entire form. – [Basil Bourque](#) Jan 12, 2019 at 0:01 
- 



You forgot about **Action-Domain-Responder** ([ADR](#)).

3



As explained in some graphics above, there's a direct relation/link between the **Model** and the **View** in MVC. An action is performed on the **Controller**, which will execute an action on the **Model**. That action in the **Model**, will **trigger a reaction** in the **View**. The **View**, is always updated when the **Model's** state changes.



Some people keep forgetting, that MVC [was created in the late 70"](#), and that the Web was only created in late 80"/early 90". MVC wasn't originally created for the Web, but for Desktop applications instead, where the Controller, Model and View would co-exist together.

Because we use web frameworks (eg.: *Laravel*) that still use the same naming conventions (*model-view-controller*), we tend to think that it must be MVC, but it's actually something else.

Instead, have a look at [Action-Domain-Responder](#). In ADR, the **Controller** gets an **Action**, which will perform an operation in the **Model/Domain**. So far, the same. The difference is, it then collects that operation's response/data, and pass it to a **Responder** (eg.: `view()`) for rendering. When a new action is requested on the

same component, the **Controller** is called again, and the cycle repeats itself. In ADR, there's **no connection** between the Model/Domain and the View (*Reponser's response*).

**Note:** Wikipedia states that "*Each ADR action, however, is represented by separate classes or closures.*". This is **not** necessarily true. Several Actions can be in the same Controller, and the pattern is still the same.

mvc

adr

model-view-controller

action-domain-responder

Share Improve this answer

edited Oct 22, 2019 at 10:24

Follow

answered Oct 22, 2019 at 10:03



Hugo Rafael Azevedo

61 ● 8

---

... that MVC was created in the late 70"... <---  
this is key! – mayo Nov 28 at 22:16

---



3



In a few words,

- In MVC, View has the UI part, which calls the controller which in turn calls the model & model in turn fires events back to view.
- In MVP, View contains UI and calls the presenter for implementation part. The presenter calls the view directly for updates to the UI part. Model which

contains business logic is called by the presenter and no interaction whatsoever with the view. So here presenter does most of the work :)

Share Improve this answer

edited Aug 25, 2020 at 15:52

Follow

answered Sep 9, 2019 at 19:31



Chinmai Kulkarni

157 ● 1 ● 4



0



## MVP

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



A presenter is acting as a supervisory role in MVP which binding View events and business logic from models.



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

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

**Pros:** The 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.

Share Improve this answer

Follow

edited Apr 1, 2020 at 3:33



Rahul

3,349 ● 2 ● 34 ● 44

answered Jan 12, 2016 at 4:50



marvelTracker

4,959 ● 4 ● 38 ● 50



**Highly active question.** Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.