Sergei Tachenov

The world is insane and quantum mechanics proves that.

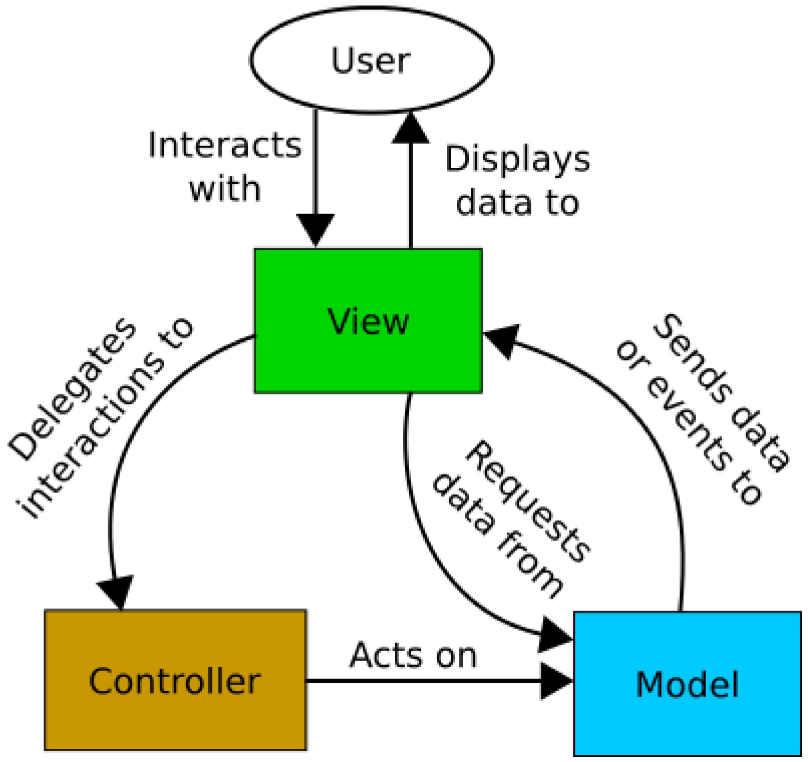
MVC, MVP and MVVM, pt. 1: The Ideas

There is a lot of confusion going on about GUI design patterns such as Model–View–Controller, Model–View–Presenter and Model–View–View Model. I'm starting this series of blog posts to share my own knowledge and experience with these patterns, hoping to clear up things a bit. I'm not going to dive deep into the history behind these patterns. Instead, I'm going to concentrate on things as they are today.

I'll start with the ideas behind these patterns. There is one single idea behind them all: separation of concerns. It's a well-known idiom, closely related to the single responsibility principle, the S part of the SOLID principles. The most clear form of it says: there should be only one reason for a class to change. Separation of concerns takes that to the architecture level: there should be only one reason for a *layer* to change. The granularity of that reason is different, though. One may say: the *Money* class should only change if the logic of working with money changes. On the architecture level one would say instead: the *view* layer should only change when appearance should change (for example, money should now be displayed using a fixed-width font). In particular, the view layer should not change if the business logic changes (money should now be calculated to 2nd digit after the decimal point) or if presentation logic changes (money should now be formatted with 1 digit after the decimal point).

Model–View–Controller

With these ideas in mind, let's go over the three patterns, starting with MVC. It's probably the most confusing of them all, and I think it's mainly because separation of concerns is not complete in MVC.



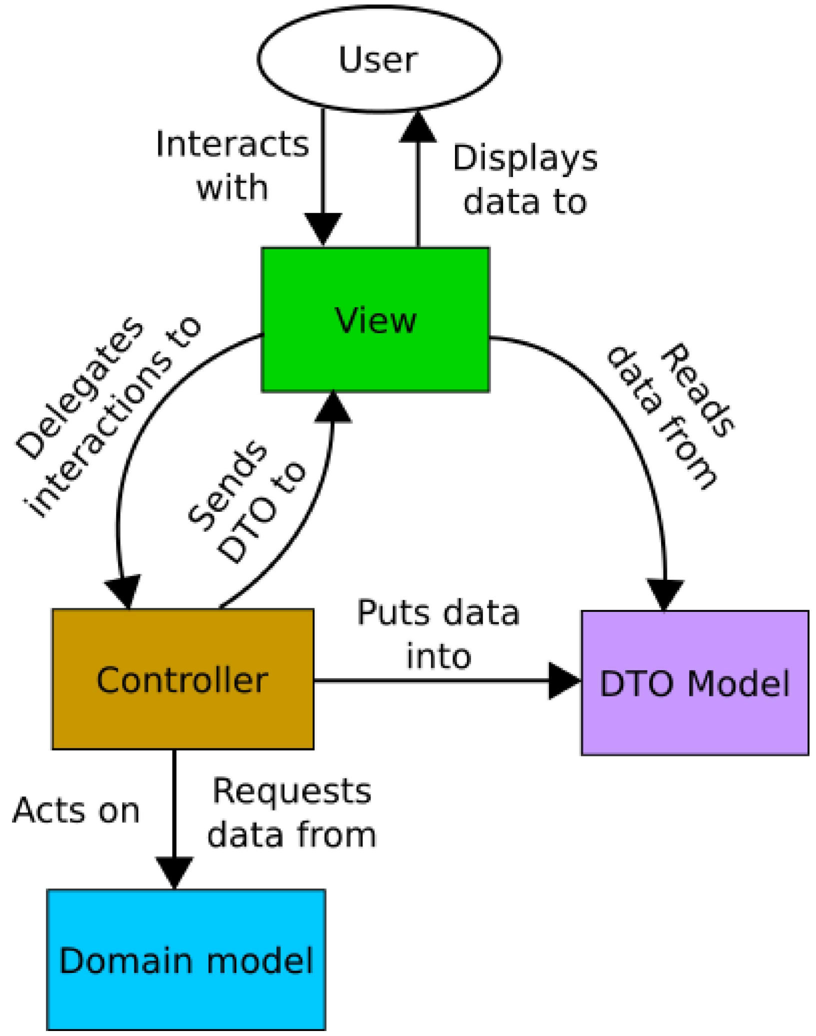
To add to the confusion, there are many variations of MVC, and there is no single agreement on what exactly the components do. The view is the easiest part: its job is to display things and receive interactions from the user. You can't really separate these two concerns: how would you separate displaying text that the user is editing and actually editing this text? There should be a single graphical component that does both of these things. You can do the next best thing, though: delegate user interactions to another component. And here is where the controller comes from.

The controller receives user interactions from the view and processes them. Depending on the interface between the view and the controller you may be able to reduce coupling between them, and that's a good thing. Suppose your view is implemented with Swing, and there is the *apply* button. Instead of making the controller implement *ActionListener,* implement it inside the view and delegate the *apply* button click to the *apply* method of the controller, which is UI-agnostic (it doesn't depend on Swing at all).

That was the easy part. But what happens next? The controller acts on the model, which contains the actual data the application works with. Then, at some point, it may be needed to display the updated data back to the user. Here is where the confusion starts. One possibility is that there is the observer pattern acting between the view and the model. In this case, the view subscribes to certain events of the model, and the model either sends the updated data to the view (the push model of the observer pattern) or just events (the pull model). In the latter case, the view needs to pull the necessary data whenever it receives the appropriate event.

Note that even though the model sends data to the view, it has no idea of its existence because of the observer pattern. This is especially important if the model is in fact the domain model, which should be isolated as much as possible. It should only communicate to the outside world through clean interfaces that belong to the model itself.

Another variation of the MVC pattern is often seen in web frameworks, such as Spring MVC. In this case, the model is a simple DTO (data transfer object), basically a hash map, easily serialized into JSON or something. The controller prepares the model and sends it to the view. Sometimes it's just a matter of passing the object inside a single process, but sometimes the model is literally sent over the wire. This is different from a typical desktop observer approach where the controller doesn't even know anything about the view. To keep this coupling loose, the controller often doesn't send the model directly to the view, but rather sends it to the framework which then picks up an appropriate view and passes the model to it.



What makes this pattern especially confusing is that the model is no longer the domain model. Rather we have two models now: the M part of the MVC pattern is the data transfer model, whereas the controller acts on the domain model (maybe indirectly through a service layer), gets back the updated data, then packs that data into a DTO and passes it to the view to display. This very idea of the data transfer model is exactly what makes this pattern so suitable for web applications, where the controller may not even know in the advance what to do with the data: you may have to wrap it into HTML and send to the browser, or maybe you serialize it into JSON and send it as a REST response.

Either way, one problem with MVC is that view is too complicated. One thing about UI is that it tends to be both heavyweight and volatile, so you usually want to keep it as clean as possible. In MVC, view doesn't only display data but it also has the responsibility of pulling that data from the model. That means the view has two reasons to change: either requirements for displaying data are changed or the representation of that data is changed. If the model is the domain model, then it's especially bad: the UI should not depend on how data is organized in the domain model. If the model is a DTO model, then it's not that bad, but it still can be changed, for example, to accommodate the need for a new view (or a REST client). Still, MVC is often the best choice for web applications, and therefore is the primary pattern of many web frameworks.

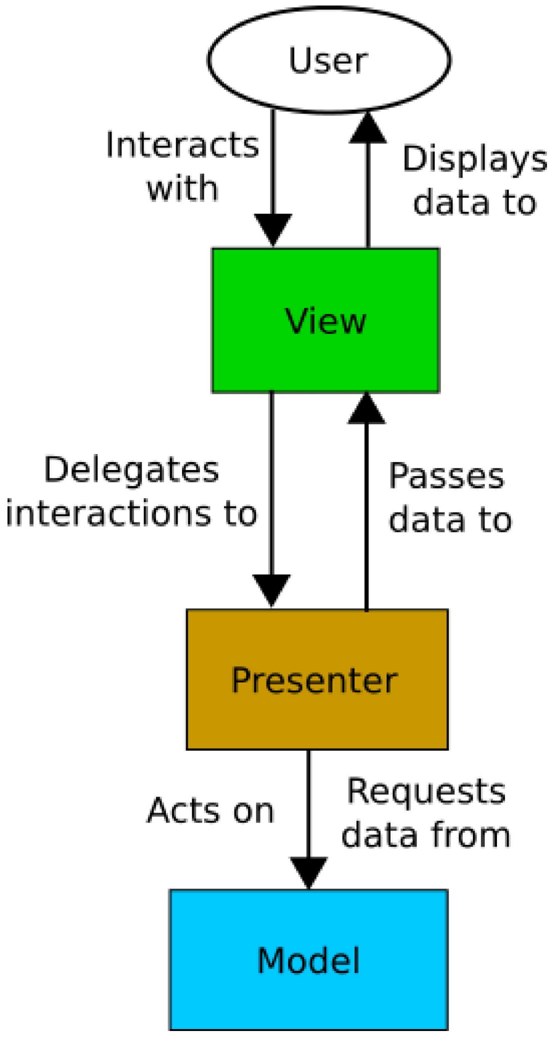
One major disadvantage of MVC is that the view is not completely devoid of logic, and therefore it can be hard to test, especially when it comes to unit tests. Another disadvantage is that you have to reimplement all that logic if you're porting your view to another tech (say, Swing to JavaFX).

Model–View–Presenter

One natural way to improve MVC is to reduce the coupling between the view and the model. If we make a rule that all interactions between the view and the model must go through the controller, then the controller becomes the single point for implementing *presentation logic.* That's what we call a *presenter.* The term *presentation logic* refers to any kind of logic that is directly related to the UI but not to the way how the components actually *look* (that's the view's responsibility). For example, we may have a rule that if a certain value exceeds a certain threshold, then it should be displayed in red color. We split this rule into three parts:

1. If a value exceeds a certain threshold, then it's too high.
2. If it's too high, then it should be displayed in a special way.
3. If it's too high, then it should be displayed in red.

The first part is the domain logic. It could be implemented, say, with an *isTooHigh* method, but it really depends on the domain. The second part is the presentation logic, and if it looks like a generalization of the third part, that's exactly what it is. The presenter knows from the model that the value is too high, and therefore, passes it to the view with some kind of *Status.TOO_HIGH* enum constant. Note that it has no idea of colors yet. It's the job of the view to actually map that constant to a color. Or maybe it could be something else than a color, like a warning sign next to the value.



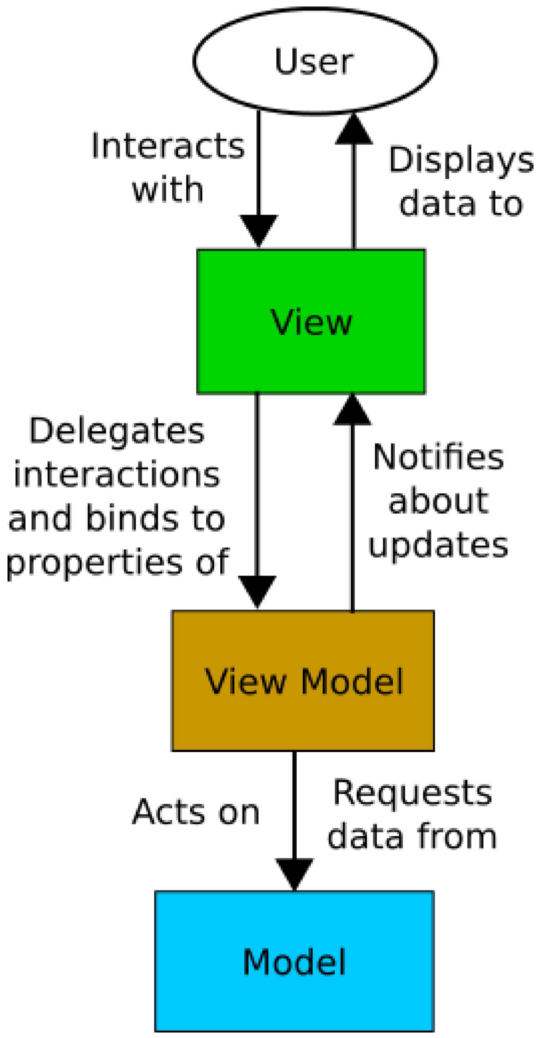
In the MVP pattern, the view is completely decoupled from the model. The presenter is something similar to the mediator pattern. In fact, if the view is implemented as a set of independent graphical components (like multiple windows), and the model also consists of multiple objects (as it almost always the case), it would be exactly the mediator pattern.

Unlike MVC, there is no observer pattern between the presenter and the view. The reason for this is that the view contains so little logic there is no place for any event handling there, except for view-specific UI events (button clicks etc.). It's the presenter's job to figure out when to update the view and do so by calling appropriate methods. These methods typically belong to an interface fully defined by the presenter, which is an excellent example of the dependency inversion principle (the D in SOLID). The presenter doesn't depend on any technologies the view uses. Well, in theory at least. For example, it would be very tricky to implement exactly the same interface with Swing, JavaFX and HTML. How do you call methods on HTML? You could have some server-side object that sends the data to the browser using AJAX or even Web Socket, I suppose, but it would be very tricky and at the same time not as powerful as MVC, where controller is free of presentation logic and therefore can be shared between views with different presentation needs (such as HTML and JSON).

The positive side is that since all presentation logic is in one place, porting to another view tech is a breeze. You just reimplement your view interface with another tech, and you got it. Well, that's at least in theory. In practice you may run into various problems. Threading, for example. Who is responsible that view methods are only called in the appropriate threads? Should the view enforce that? Probably yes, because the presenter has no way of figuring out which thread is right if it has no idea what GUI framework is used in the first place. But that imposes additional burden on the view. But still, MVP is probably as close as you can get to the perfection of total independence from the GUI framework used.

The bad news is that presenter now contains a lot of boilerplate code. It was a part of the view before, so it's not like it became any worse than it was with MVC, but still it's always a nice idea to get rid of as much boilerplate code as possible. That's where MVVM comes into the picture.

Model–View–View Model



MVVM is basically the same thing as MVP, except for one major difference. In MVP, the view only delegates user interactions to the presenter. Whenever the feedback is needed, it's the presenter who takes action. It does it by literally calling methods on the view such as *displayFilesList(files), setApplyEnabled(true), setConnectionStatus(ConnectionStatus.GOOD)* and so on. That's boilerplate code. With MVVM, the presenter becomes the view model, that is, a model that provides access to the ready-to-display data through the observer pattern, much like in desktop MVC. Except that now the view model can really prepare that data for display by filtering, sorting, formatting etc. So whatever presentation logic was in the view in MVC, it's now in the view model. And while in MVP the presenter pushed that data from to the view, in MVVM the view pulls that data from the view model. This sounds like we're adding responsibility to the view, and that's a Bad Thing, right? Well, to a certain extent, yes. But the point is, this responsibility is typically almost entirely implemented by the framework. This is done through data binding, where you just specify that this component should display that property of the view model, and that's basically it.

When your framework doesn't support data binding, it's usually a bad idea to use MVVM because you'll essentially be moving the boilerplate code from the presenter to the view, which is indeed a Bad Thing. And even if you have data binding, it's usually not that simple. Sometimes you have complicated structures to bind. Sometimes the order of updates is important and you have race conditions in your UI. Sometimes you have values of custom types that are tricky to display directly, you need to employ some sort of converters for that.

The good part is that with MVVM you typically only have problems when you have a non-standard situation. For most cases, it really decreases the amount of boilerplate code and displaying a person's name in a text field becomes as simple as writing *Text="{Binding Person.Name}"* in XAML.

Moreover, delegating user interactions is often implemented with data binding too. Well, as I say "often", I really can't think of any other MVVM implementation than .Net/WPF, so I guess it's 100% of all cases, even though there is only one case in total! Nevertheless, using the command pattern, we can expose possible interactions as properties of the view model. The view then binds to them and executes appropriate commands when the user does something. One big advantage of it is that we can easily change these commands dynamically and the view will automatically update its interactions.

When choosing between MVVM and MVP, it's important to consider several factors:

- If your framework doesn't support data binding, MVVM is probably a bad idea.
- If it does, then how likely that you'll want to switch UIs? How painful is it likely to be?
- How difficult it would be to port your application from MVVM to MVP or MVC or vice versa?

All things being equal, it's often the case that reimplementing the view interface for a new framework in MVP pattern is about as hard as switching from MVVM to MVP or whatever. In this case, it's probably worth to use MVVM if that's the thing with your framework. The same really goes about using MVC. When your framework offers you MVC, you probably don't want to force yourself to use MVP instead unless you really plan to switch frameworks and design for it beforehand. Say, you're using Swing right now, but you know you'll have to move to JavaFX within 5 years.

One last thing to note is that it is possible to combine these patterns, although in most cases it's likely to lead to over-engineering. For example, if your framework forces you to use MVC, you can really turn your controller into a view model, and then consider the whole view–view model part to be just a view for the MVP pattern. So when user does something, the view delegates that to the controller, which immediately delegates to the presenter. When the presenter gets updated data from the domain model, it sends that data to the controller (using a clean interface), which then stores it locally and fires an event to the actual view which pulls that data to actually display it. Sounds crazy enough as it is, doesn't it? But sometimes it may be worth it, only experience can tell you. It's probably best to start with the simplest thing possible, and complicate things only when you actually need it.

That's it for now. I plan later to demonstrate all three patterns with a simple application. I'll probably use Java for that, even though implementing MVVM would be tricky. But there is some limited data binding in Java, so it could actually work, if only for demonstration purposes.

This entry was posted in Uncategorized on 2016-09-30 [https://web.archive.org/web/20210801011438/https://www.tachenov.name/2016/09/30/208/] .