Software Design and Algorithms



MV\* Patterns

EPAM Systems Inc.

Learn & Development

Software Design and Algorithms

[www.epam.com](http://www.epam.com/)

Table of Contents

[MV\* Patterns 1](#_Toc64584466)

[1. Introduction 3](#_Toc64584467)

[1.1. Good architecture vs design smells 3](#_Toc64584468)

[1.2. Why do we need structural mv\* patters? 4](#_Toc64584469)

[2. mvc 6](#_Toc64584470)

[2.1. passive/active model 7](#_Toc64584471)

[2.2. user interaction with the view 8](#_Toc64584472)

[2.3. mvc in web 9](#_Toc64584473)

[2.3.1. Model 9](#_Toc64584474)

[2.3.2. View 10](#_Toc64584475)

[2.3.3. Controller 11](#_Toc64584476)

[2.4. mvc in architecture 11](#_Toc64584477)

[2.5. original reenskaug’s scheme 14](#_Toc64584478)

[2.6. domain model facade. provider dicatates interace 15](#_Toc64584479)

[2.7. how to apply 16](#_Toc64584480)

[2.8. summary 17](#_Toc64584481)

[3. Mvp 19](#_Toc64584482)

[3.1. p for presenter 19](#_Toc64584483)

[3.2. passive view 20](#_Toc64584484)

[3.3. supervising controller 21](#_Toc64584485)

[3.4. summary 21](#_Toc64584486)

[4. mvvm 22](#_Toc64584487)

[4.1. model 22](#_Toc64584488)

[4.2. view 23](#_Toc64584489)

[4.3. view-Model 23](#_Toc64584490)

[4.4. summary 24](#_Toc64584491)

[Conclusion 25](#_Toc64584492)

# Introduction

Every program that needs to interact with a user requires a user interface. Through years of evolution of programming engineers learned some best practices represented in MV design patterns. Design patterns are reusable solutions to commonly occurring problems in software design.

This lecture discusses existing MV patterns classified in three main families: Model-View-Controller (MVC), Model-View-View Model (MVVM), and Model-View-Presenter (MVP). All of them are intended to address the needs of interactive applications by separating the concerns assigned to different components within their respective architectures. While similar, each of these patterns differs slightly in their motivations and applicability to various design goals.

## Good architecture vs design smells

Before we start talking about structural patterns, we need to clarify some important things about architectures because these patterns were created from analyzing of a vast number of different architectures and designed to build a good one.

So, what is a good architecture? How can we define that the architecture we are considering for our application is good or bad?

Analyzing the history of software development, we can say that good architecture allows us to edit, add or remove some features without being forced to change other modules or the entire system at all, so it can be done fast.

Basing on experience of thousands of developers and projects we can say that good architecture has these characteristics:

* Efficiency: all features have to work as it required and handle errors properly.
* Flexibility: the ability of software to change easily in response to different user and system requirements.
* Extensibility: the ability to add new functionality easily without need to update many other parts of application.
* Scalable development process: increasing the amount of developers does not require a change in the design.
* Testability: modules of your app should be easy to test.
* Reusability: your code does not have duplicates.
* Maintainability: well structured, readable, and understandable code. Maintainability.

On the other hand, we have some characteristics which can describe bad architecture. They can be classified as code smeels, as we discussed previously. When creating architecture, you have to avoid them:

* Rigidity: the design is hard to change.
* Fragility: The design is easy to break.
* Immobility: The design is hard to reuse.
* Viscosity: it is hard to do the right thing.
* Needless complexity: overdesign.

If you can describe you design with at least one of these points, you need to review it and possibly change it.

## Why do we need structural mv\* patters?

In spite of all these characteristics the main thing about building architectures is reducing complexity. And to reduce complexity, nothing but division into parts has yet been invented. This is sometimes called the divide et impera principle, but essentially it is a hierarchical decomposition.

"Although MV\* patterns have quite a few differences between themselves, their goals are similar: to separate the UI code (View) from the logic code (Presenter, Controller, ViewModel, etc.) and data processing code (Model). This allows each of this pattern parts to be developed independently. For example, you can change the look and style of your application without affecting business logic and data. Also dividing your application into independent parts allows you to simplify writing tests by decreasing its dependencies. As it is shown in Figure 1, on the left we have an example of a system. Modules in this system do not have a clear separation between each other so changing even some small detail in one module may affect work of second one, and changing second module will break third one and etc. This makes improvements to our app expensive and hard to do.

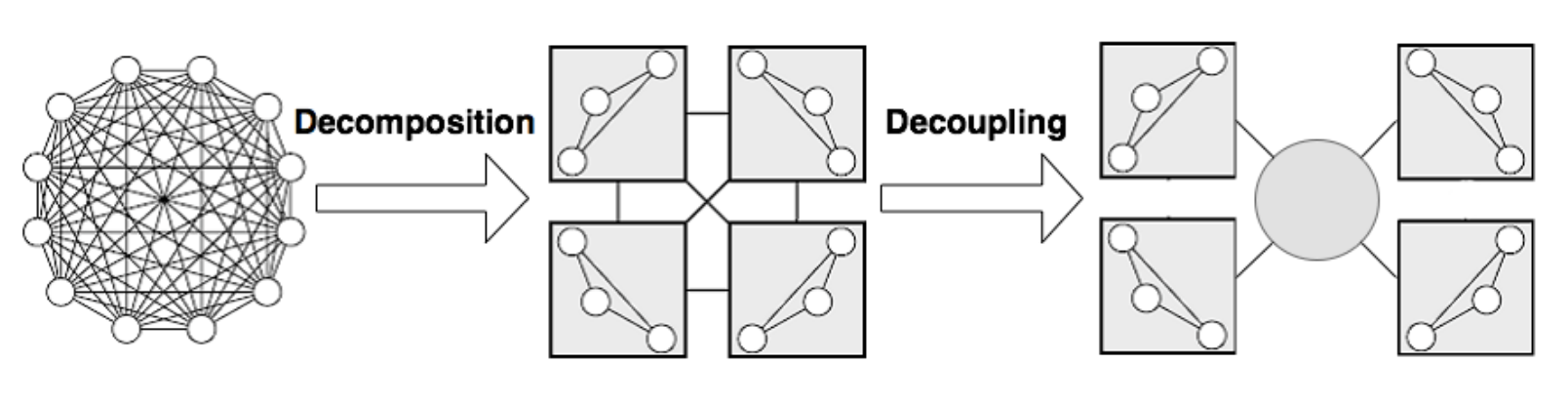


Figure 1

So, this is what MV\* patterns offer: efficient way to decompose logic of applications with user interface(s).

# mvc

MVC is a first pattern we are going to look at. MVC separates the representation of the application domain (Model) from the display of the application's state (View) and user interaction control (Controller). Each of these components is designed to handle some aspects of your app. Model is responsible for handling your data and operating it. This is where the business logic of your application should go. View is what your users see and interact with. Last but not least, controller is a part which makes view and model communicate with each other. Splitting your app into these three parts allows you to reuse them or replace easily without changing other parts.

As [this](http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/) source says:

The MVC pattern was originally conceived in 1978-79 by Trygve Reenskaug and had the primary goal of providing an interface for users to manipulate multiple views of data as if working with real world entities. The pattern was first referred to as the Thing-Model-View-Editor before Reenskaug and his colleges settled on the name Model-View-Controller (1). A modified version of Dr. Reenskaug’s design was later implemented as part of the Xerox PARC Smalltalk-80 class library. A description of this implementation can be found in the work: Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC).

MVC became popular and a lot of sources about it exist now. The problem is, a lot of sources describe it in a wrong way, or oversimplify the pattern. This results in confusion between developers. In a more advanced version, MVC is called a composite pattern, that is, a combination of several patterns that work together to implement complex applications (usually Observer, Strategy and Composite are mentioned here). And only a few people understand that

MVC is primarily a set of architectural ideas / principles / approaches that can be implemented in different ways using different patterns.

## passive/active model

MVC schemas can be divided into two types: passive and active view (Figure 2). Your view needs to get the data from model and it can be achieved in two ways. In first case view knows about model and it gets the data directly from it. In second case, data from model is provided by controller so view knows nothing about the model, which makes it more modular. Crossed line between Controller and View on “Active View” schema means that this relation is not required.

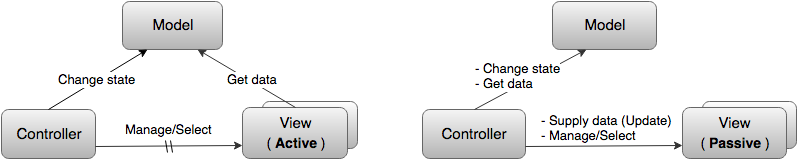


Figure 2

There can be several views - they can display the same data in different ways, for example, in the form of a table or a graph, or they can be responsible for displaying different parts of the data from the Model. The Controller is the most controversial component because it may take different functions like handling view events or be responsible for thansferring data between View and Model. Nevertheless, it is common that the Controller is always aware of the Model and can change it (usually as a result of user actions). It can also manage the View / Views (especially if there are several of them) and accordingly know about the Views, but this is not necessary.

A model, like a View, can also be Passive or Active (Figure 3). The Passive Model has no effect on either the View or the Controller. In this case, all changes to the Model are monitored by the Controller and it is also responsible for redrawing the View when necessary.

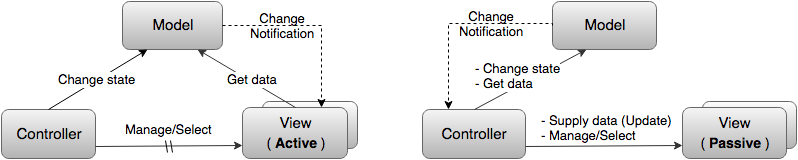


Figure 3

But usually, by MVC they mean the variant with the Active Model. The "Active Model" notifies that changes have occurred in it. And it does this through the Observer template, sending notifications of changes to all its "subscribers". The "Active View" subscribes to these messages itself and thus knows when to re-read the data it needs from the model and update itself. In the case of a Passive View, the Subscriber is the Controller, which then updates the View. The Observer pattern allows the Model, on the one hand, to inform the View or the Controller that changes have occurred in it, and on the other hand, in fact, "not know" anything about them (except that they implement a certain specified "subscriber" interface) and thus remain independent. This is called loose coupling and is considered the second key point of MVC.

## user interaction with the view

As we already know, MVC is a pattern for development of applications with user interface so it means that on out schema user is presented always. User interacts with the application and calls some model’s functions by this interaction.

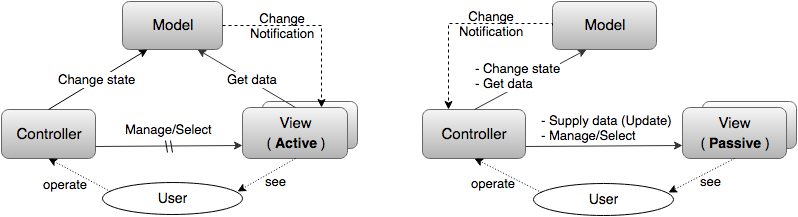


Figure 4

In first scheme (Figure 4) the user controls the application through the Controller, and the View serves solely to display information about the Model, and the user only sees it. In this case by View we can some HTML template engine for example.

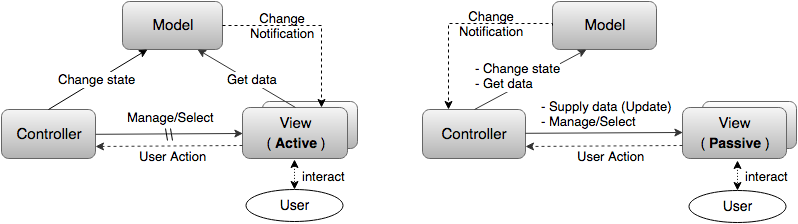


Figure 5

In second one user interacts only with the View (Figure 5). That is, the View not only reflects the Model, but also receives user commands and transfers them to the Controller. In this case, another connection is formed between the View and the Controller:

* **direct** (the View knows about the Controller and directly transmits information) or, most often.
* **weakened** (the View simply sends information about user actions to all interested subscribers and the Controller subscribes to this mailing).

## mvc in web

MVC is commonly used in Web development. In context of it we can say that Model can be represented by database. Your web server which is interpreted as Controller. Sometimes it is also said that the controller is the center for processing all requests and making decisions, as well as an intermediate layer that provides communication between the model and the view. View is the user interface. Moreover, the interface in this case, as a rule, is understood exclusively as "design", just a set of graphic elements.

Let us check every component usage in Web development. This way of mapping MVC to Web is simplified too much and we will discuss it in next section.

### Model

The Model is responsible for getting and manipulating the data, so it is the brain of the application. Usually, it interacts with some kind of database. This could be a relational database, like MySQL, or a NoSQL database like a MongoDB. It really does not matter and in many frameworks that support multiple databases, the model code will stay the exact same.

It is just the database driver that needs to change and it does not even have to be a database that works with it. It could be a simple file. So, you could just have your model interact with a JSON file and pull data from that.

### View

Next, we have the view and you can guess what that takes care of. That is the actual view of the application, so it is user interface, it is what the user sees and how they interact with the application.

So in web, the view usually consists of HTML and CSS along with dynamic values from the controller. So, the controller communicates with the view as well as the model. Now, depending on which framework you use, the template engine may vary.

The template engine is what allows dynamic data. If we have straight HTML, we cannot output variables, we cannot use logic, select an if statement, etc., but, with template engines, we can do that stuff right in the view or right in the template.

So, some examples of template engines would be Handlebars.js and Dust.js. Then, for Ruby on Rails, we have ERB, which is embedded. For Ruby, you can also use Haml and then with Python you have a Flask. There is a lot more of them out there and some do more than others, but you have a lot to choose from, especially with JavaScript.

### Controller

So, finally, we have the controller and the controller takes in user input, so this could be from the user visiting a page or clicking a link which makes a get request or submitting a form which makes a post request and we also have delete requests or put requests for updating. And these cannot be made directly from the browser, you can only do a get or a post but we do have HTTP clients that are at times built in with the framework, that can do that.

Now the controller acts as kind of an agent between the model and the view. The controller will ask the model to get some data from a database and then the controller will take that data, load a view, and pass that data into it. Then, from there, the template engine takes over and can do some logic, output variables, and things like that.

The controller can also load a view without passing it data, so just a plain web page with HTML and CSS, no actual templating logic.

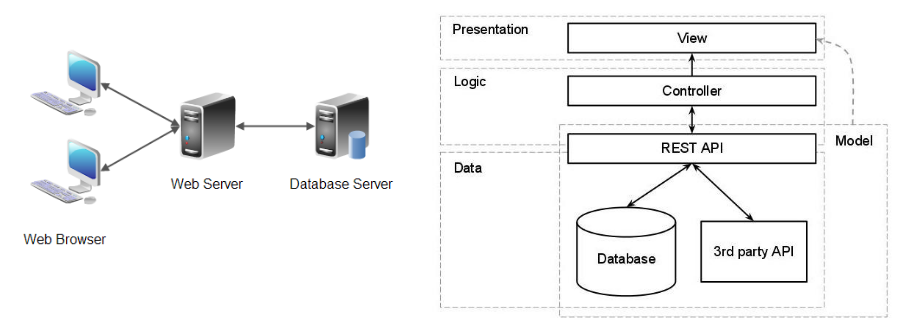


Figure 6

## mvc in architecture

The way of thinking understanding MVC from previous section might work but it’s not entirely correct because:

* **Model independence is central to MVC**. If the Model is thin, that is, it contains only data, then the possibility of its independent development makes little sense. Accordingly, with this approach, MVC itself loses its meaning.
* All the business logic of the application, that is, most of the code, is concentrated in the Controller, and this is despite the fact that **the Controller is the most dependent part in MVC** - in general, it depends on both the Model and the View. In well-designed applications, they try to do exactly the opposite - the most dependent parts should be minimal, not maximal.
* Since, as a rule, not only the Model is thin, but also the View is thin (Dumb GUI, Dumb View), then, as a result, in addition to all the business logic of the application, the controller also places the logic for managing the user interface. That is, instead of separating business logic and presentation logic, this approach results in their mixing.

So, it is better to take MVC not like a pattern, but a set of architectural principles and ideas used in the construction of custom information systems (usually complex). The essence of such principles is as follows: **a complex system must be divided into modules**. Moreover, it is advisable to do the decomposition hierarchically, and the modules into which the system is divided should be, if possible, independent, or weakly coupled (**Low coupling**), just like it is shown on example on Figure 7. All presented modules are separated from each other so it is easier to test them and develop independently.

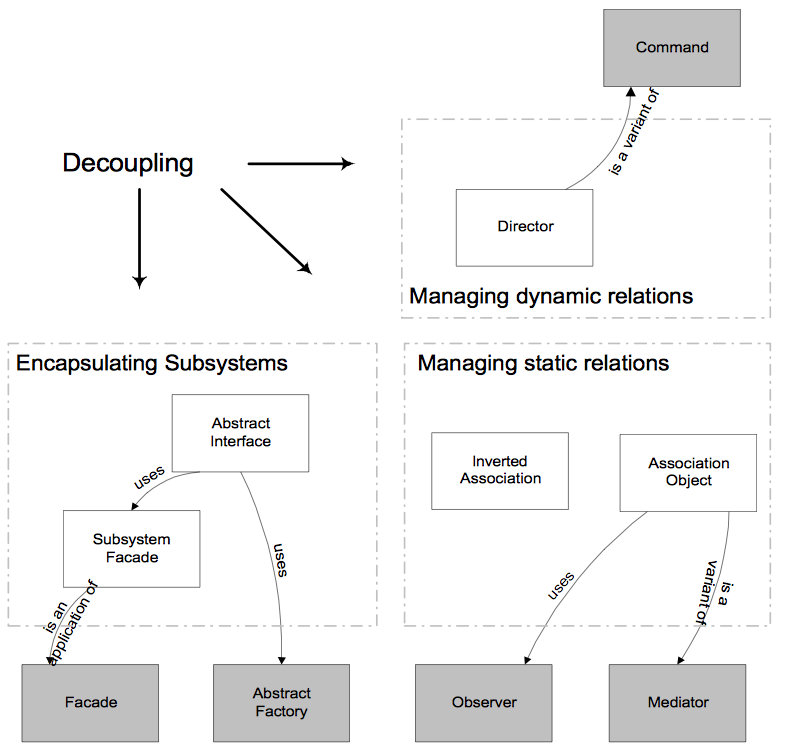


Figure 7

Basically, MVC is based on three simple ideas:

* **Separate Model and UI into modules**.

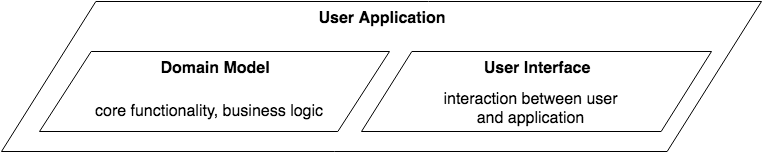


Figure 8

Any user application can be divided into two modules - one of which provides the functionality of its application, its logic, and the second is responsible for interaction with the user. This allows us to develop a domain model (domain model) that contains the business logic of the system. The task of interaction with use is carried out in a separate module - the user interface and can also be solved independently. It is the domain mode that is considered the Model (M) in "architectural MVC" (hence the term).

* **Decouple Model and UI. Independent Models and User Interface synchronized via Observer pattern. This allows Model to be independent and know nothing about its subscribers.**

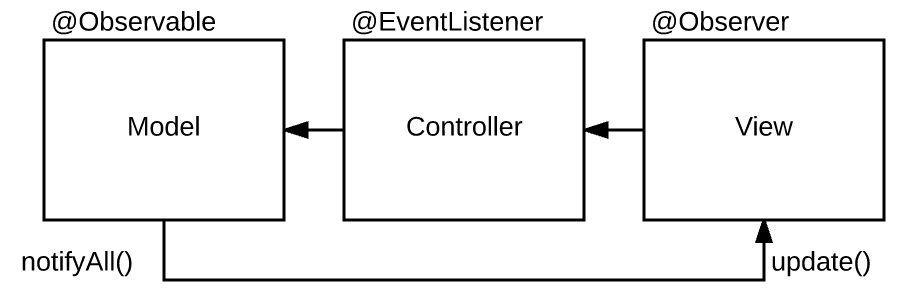


Figure 9

* **Separate UI into View and Controller.**

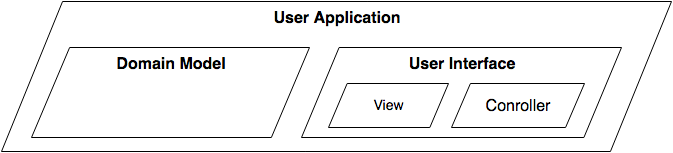


Figure 10

Looking at architecture of your application in this way allows you to split development of your app into separate parts and scale or change them easily when needed.

## original reenskaug’s scheme

In general, is the Model one object or multiple objects? And is the Model in the "MVC Scheme" really the same as the domain model that describes the domain and business logic of the application? When it was said that the system should be divided into modules that are loosely connected to each other, we did not mention the main rule - modules should interact with each other only at the level of abstract interfaces (Dependency Inversion Principle). A module interface is usually implemented by a special object - Facade. Thus, the key idea of ​​MVC is really that the user application is divided into two modules - one of which models the domain and implements business logic (domain model), and the second is responsible for user interaction (user interface). But at the same time, the Model in the "MVC scheme" is not at all identical to the domain model (which can be arbitrarily complex and consist of many objects), but is just its interface and facade.

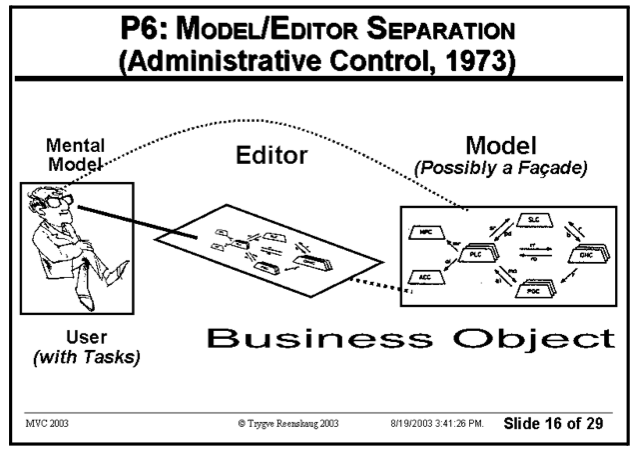


Figure 11

Figure 11 is a slide from original Reenskaug’s presentation. Check out the text in parentheses “Possibly a Façade.” So, it does not matter for the MVC is our Model is a real domain model or just a facade which encapsulates more complex logic.

## domain model facade. provider dictates interface

Traditionally, in client-server applications, the server is considered the main one. He provides services and decides in what form it should be implemented. So, the interface and facade are usually defined from the server's point of view. And clients adapt to this specified format.

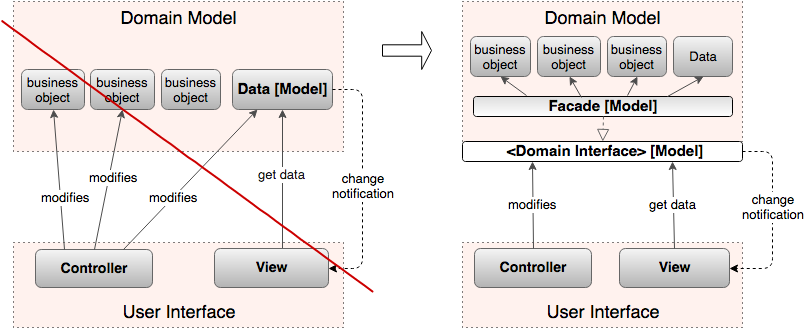


Figure 12

In practice, it is not a server-oriented architecture that is more adequate, but a client-oriented one. The client-centric approach is much better suited to the Interface Segregation Principle, as it uses many thin Required Interfaces instead of one thick Provided Interface.

Due to the fact that the facade present in the original MVC has been "lost", its role is often taken over by the Controller. This is the origin of the idea that the Controller is "between the Model and the View", serves as a glue between them and provides the data the View needs.

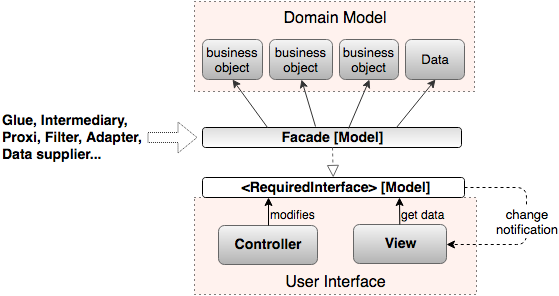


Figure 13

## how to apply

Now we can check how MVC can be implemented with some programming language, Typescript in our case.

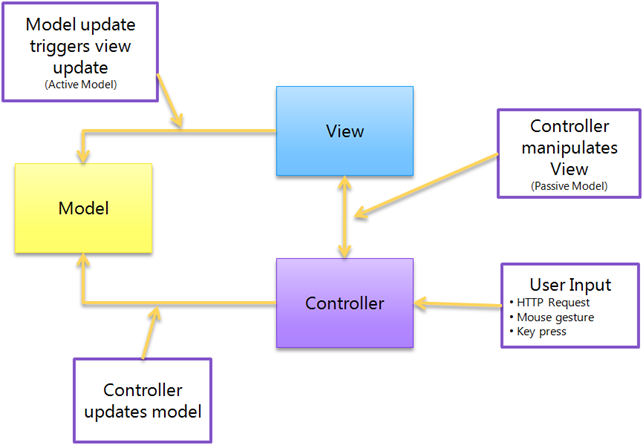


Figure 14

As usual we have three main components: Model, View and Controller.

Model (domain model in our case) contains data and some methods to interact with it. In our case this data is stored in some database. But keep in mind that data can have different origins like some API’s or other services.

View is a layer of displaying data on customer's screen. Without any logic, usually provided by frameworks like Angular, Vue or done via HTML-templates. In our case it is a HTML DOM tree.

Controller handles user interactions with View and updates Model and View. It is responsible for making decisions like what data and when should be updated or whether it needs to be updated at all.

class Controller {  
 private readonly \_view: IView;  
  
 constructor (view: IView) {  
 this.\_view = view;  
 }  
  
 public async initialize () {  
 const response = await getResponseFromSomewhere();  
  
 if (Number.isInteger(response.personId)) {  
 this.showPerson(response.personId);  
 } else {  
 this.showError('The ID was missing.');  
 }  
 }  
  
 private showPerson(id: number): void {  
 const repository = new Repository();  
 const model = repository.getModel(id);  
  
 this.\_view.showPerson(model);  
 }  
  
 private showError(s: string): void {  
 this.\_view.showError(s);  
 }  
}

It should be noted that the Controller receives input events directly and not through the View. The controller interprets user input from the keyboard or mouse, and sends commands to the model and view to make the appropriate changes.

## summary

So, what advantages MVC gives us?

* First it is easier overall maintenance. When updates need to be made to the application it is clear whether the changes are data-centric, meaning changes to models and controllers, or merely visual, meaning changes to views.
* Decoupling of models and views means that it is significantly more straight-forward to write unit tests for business logic.
* Duplication of low-level model and controller code (i.e., what we may have been using instead) is eliminated across the application.
* Depending on the size of the application and separation of roles, this modularity allows developers responsible for core logic and developers working on the user interfaces to work in parallel.

Today’s development environments have come a long way in shielding developers from such complexities, making divisions between device input and output at the application level unnecessary. Within many development environments, the original goals of the Model-View-Controller pattern can be carried out today by merely separating an application’s Forms and associated Controls from its domain model.

# Mvp

MVP (Model-View-Presenter) is a product of evolution of MVC. In previously discussed pattern View is responsible only for displaying data and Controller handles all of View’s interactions and events. During the development of Web, we have some elements which can handle events and user input by itself. Now controller does not need to be responsible for handling of this operations and here we have Presenter. In fact, MVP is a more modern way of building apps with user interfaces.

## p for presenter

In fact, Presenter is the business logic component for the View. Unlike MVC, calls from the View are delegated to the Presenter, which is separate from the View and communicates with it through an interface.

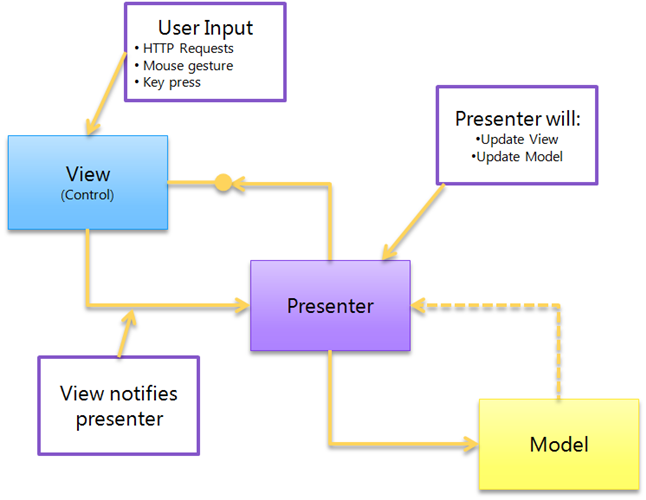


Figure 15

In MVP we have three components:

* **Model.** This is the data of your application, the logic of their receipt and storage. Often it is based on a database or on the results from web services. In some cases, you will need to adapt, modify or expand it before using it in View.
* **View.** Usually is a form with widgets. The events of View should be sent to the presenter.
* **Presenter.** The presenter contains all the user interface logic and is responsible for synchronizing the model and the view. When the view notifies presenter that the user interacts with the system (for example, presses a button), the presenter decides to update the model and synchronizes all changes between the model and the view.

## passive view

There are two types of this pattern: Passive View and Supervising Controller.

In Passive view MVP variant, the view knows nothing about the model, but instead provides simple properties for all the information that needs to be displayed on the screen. The presenter will read information from the model and update properties in the View.

Models can trigger events; Presenters can subscribe to them if they wish to update the View. In this passive architecture, we do not have the concept of direct data binding. Views provide setters that Presenters can use to set data.

class PersonalDataView implements IPersonalDataView {  
 get userName(): string {  
 const input: HTMLInputElement = document.getElementById('userInput');  
  
 return input.value;  
 }  
  
 set userName(value: string) {  
 const input: HTMLInputElement = document.getElementById('userInput');  
  
 input.value = value;  
 }  
}

The benefit of this change from MVC is that it improves the testability of our application and provides a clearer separation between view and model.

While a typical Passive View implementation involves implementing an interface with setters and getters, there are options that involve the use of events that can unbind the Presenter View.

## supervising controller

In this MVP variant, the view is aware of the model and is responsible for binding data to the display. This makes the communication between the Presenter and the View more concise.

class PersonalDataView implements IPersonalDataView {  
 public setPersonalData(data: PersonalData): void {  
 const input: HTMLInputElement = document.getElementById('userInput');  
  
 input.value = data.firstName;  
 }  
  
 public updatePersonalData(data: PersonalData): void {  
 const input: HTMLInputElement = document.getElementById('userInput');  
  
 data.firstName = input.value;  
 }  
}

## summary

So, what does MVP gives us?

It makes debugging easier in Applications — MVP enforces three different layers of abstractions which makes it easier to debug your applications. Moreover, since business logic is completely decoupled from View, it is easier to perform unit testing while developing your application.

Also, it enforces better separation of Concerns — MVP does an excellent job of separating out your business logic and persistence logic out of your classes which in turn better enforce good separation of concerns.

And finally, it improves code re-usability — In MVP, the code can be better reused since you can have multiple presenters controlling your Views. This is more important as you do not want to rely on a single presenter to control your different Views.

# Mvvm

MVVM stands for Model-View-ViewModel. From one point of view, it can be considered as evolution of MVC, but from another it is fair to say that MVVM is fundamentally different. It was implemented in 2005 by John Grossman. The main idea behind this pattern is that it is trying to fully separate development of view and UI from development of Model and business logic.

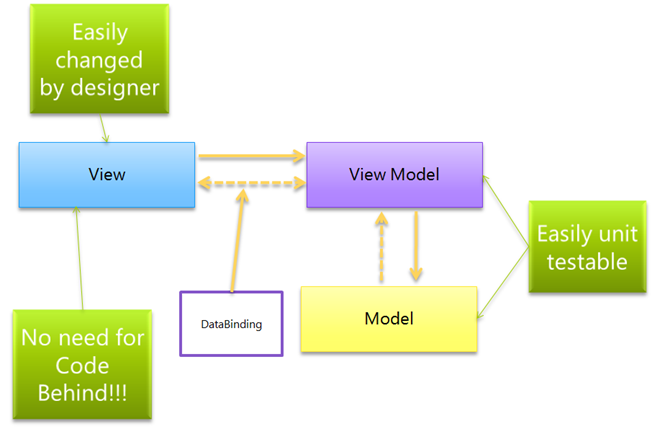


Figure 16

## model

 const Todo = function (content, done) {  
 this.content = ko.observable(content);  
 this.done = ko.observable(done);  
 this.editing = ko.observable(false);  
}

The ViewModel cannot communicate with the View directly. Instead, it presents easily bindable properties and methods as commands. The View can bind to these properties to get information from the ViewModel and call commands (methods) on it. This does not require the View to be aware of the ViewModel.

As with other members of the MV \* family, a model in MVVM represents data or domain information that our application will work with.

## view

const aViewModel = {  
 contactName: ko.observable('John')  
}  
  
const template = `  
 <input id="source" data-bind="value: contactName, valueUpdate: "keyup" />  
   
 <div data-bind="visible: contactName().length > 10">  
 You have a really long name!  
 </div>  
`

As with MVC, the View is the only part of the application that users interact with. This is some kind of interactive interface that represents the state of the ViewModel.

The Active View in MVVM contains databindings, events and behavior, and this requires an understanding of the ViewModel. The View is still responsible for handling events from the ViewModel here.

It is important to remember that the View is not responsible for the state, it just synchronizes with the ViewModel.

## view-Model

The ViewModel can be thought of as a specialized controller that acts as a data converter. It converts model data to View data by passing commands from View to Model.

The ViewModel can also provide methods to help maintain the state of the View, update the model based on actions from the View, and trigger events on the View.

As a result, the ViewModel is behind the UI layer. It provides the data from the model that the View needs, and can be thought of as the source that the Views access for both data and actions.

const ViewModel = function (todos) {  
 // map array of passed in todos to and observableArray of Todo objects  
 this.todos = ko.observableArray(  
 todos.map(todo => new Todo(todo.content, todo.done))  
 );  
  
 // store the new todo value being entered  
 this.current = ko.observable();  
  
 // add a new todo, when enter key is pressed  
 this.add = (data, event) => {  
 this.todos.push(new Todo(this.current));  
 }  
}

## summary

MVVM facilitates easier parallel development of a UI and the building blocks that power it.

MVVM abstracts the View and thus reduces the quantity of business logic (or glue) required in the code behind it.

The ViewModel can be easier to unit test than in the case of event-driven code.

The ViewModel (being more Model than View) can be tested without concerns of UI automation and interaction.

Conclusion

In **MVC**, the View sits on top of our architecture with the controller beside it. Models sit below the controller, so our Views know about our controllers and controllers know about Models. Here, our Views have direct access to Models. Exposing the complete Model to the View, however, may have security and performance costs, depending on the complexity of our application. MVVM attempts to avoid these issues.

In **MVP**, the role of the controller is replaced with a Presenter. Presenters sit at the same level as views, listening to events from both the View and model, and mediating the actions between them. Unlike MVVM, there is not a mechanism for binding Views to ViewModels, so we instead rely on each View implementing an interface allowing the Presenter to interact with the View.

**MVVM** consequently allows us to create View-specific subsets of a Model, which can contain state and logic information, avoiding the need to expose the entire Model to a View. Unlike MVP’s Presenter, a ViewModel is not required to reference a View. The View can bind to properties on the ViewModel, which in turn expose data contained in Models to the View. As we have mentioned, the abstraction of the View means there is less logic required in the code behind it.