**Reporter: Nguyen Duong Thu Thuy**

**Software Architecture**

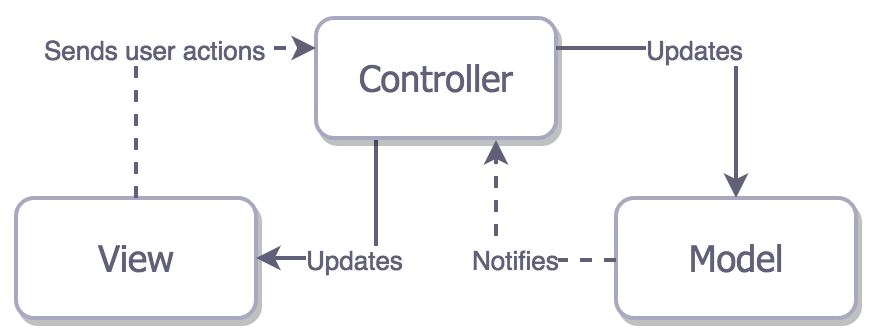
**MVC, MVP, MVVC, VIPER**

* Features of a good architecture
* Balanced distribution of responsibilities among entities with strict roles
* Testability usually comes from the first feature
* Ease of use and a low maintenance cost

1. **MVC (Model – View – Controller)**

The Model-View-Controller concept describes 3 components:

* **Model**, a wrapper of data
* **View**, a representation of a user interface (UI)
* **Controller**, an intermediary between the Model and the View



Every component has a distinct role:

* The Model encapsulates a particular set of data, and contains logic to manipulate that data. When you think about accounting software, an **Invoice** is a model. When you think of a Twitter app, a **Tweet** is a model.
* The View is an object that the user can see, in a user interface (UI). In the accounting software, a UILabel that **displays the invoice address** is a **view**. In a Twitter app, the TweetView is a **view that displays a tweet**.
* The Controller controls all logic that goes between the View and the Model. It transports **messages** between the **View and the Model**, and vice versa.

Why we should use MVC?

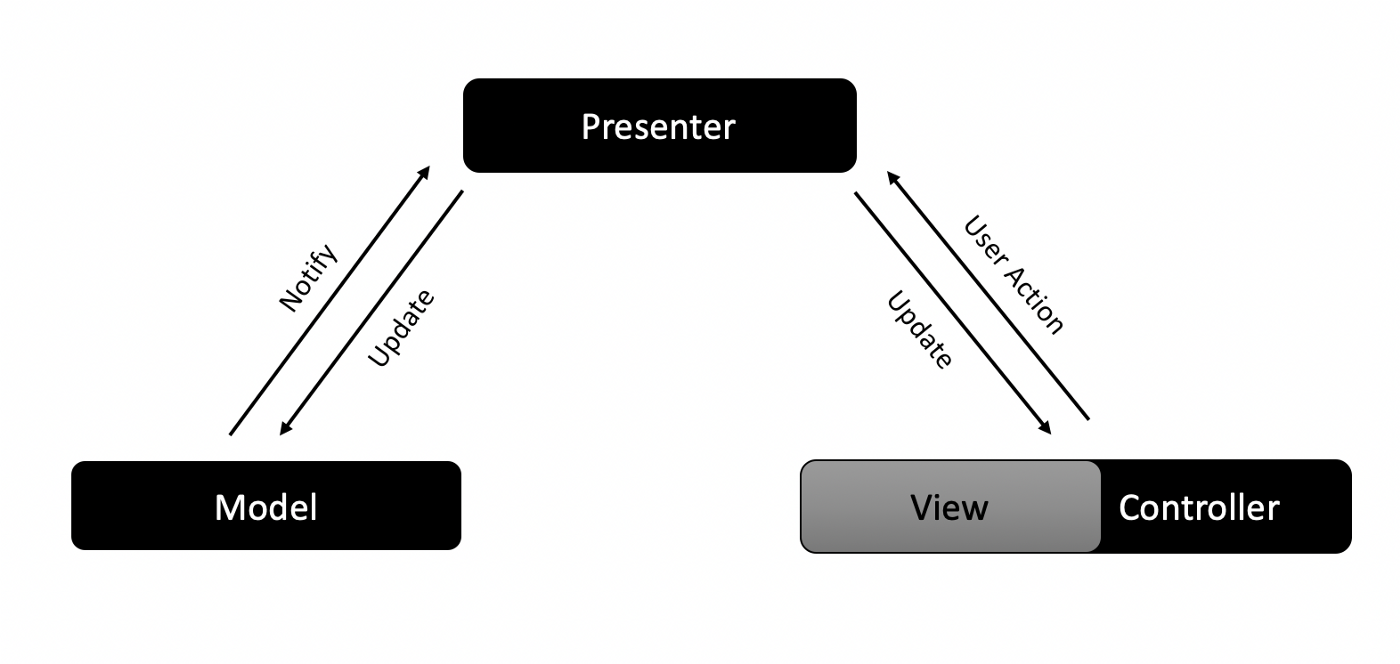
* This architecture helps keeping your app maintainable, keeps your code from becoming a big pile of chaos
* Most devs know and understand MVC

Some drawbacks of using MVC:

* Weak separation of concerns, the controller handle every thing here almost, it handles the business logic and the presentation logic, updating UI, applying some animations.
* Hard to unit test because business logic, presentation logic and UIKit members are mixed in the controller. Testable classes shouldn’t depend on any UIKit member
* Can not scale up easily
* Hard to maintain, when the project get growing the controller will be chaotic
* MVC seems like a very bad pattern to choose for your app.
* MVC is the best architectural pattern in terms of the speed of the development.

1. **MVP (Model – View – Presenter)**

The MVP pattern is the derived from Model-View-Controller pattern, but it has some discrepancies. the MVP pattern has a new layer which called Presenter, and here is the main key of MVP:



* The **View** here is the same as the view in **Model-View-Controller** pattern, except that the View should not interact with the Model directly, the View can **interact only** with the **presenter**
* The **Model** is also the same as the model in **Model-View-Controller** pattern, but the Model also can not interact directly with the View, it should **interact only** with the **Presenter**
* The **Presenter** is the new layer introduced by Model-View-Presenter pattern, as illustrated in picture above, the Presenter is an intermediate layer that **handle the communication** between **View** and **Model**. it solved a lot of MVC problems like **testability, maintainability, scalability**.

However, MVP has some problems:

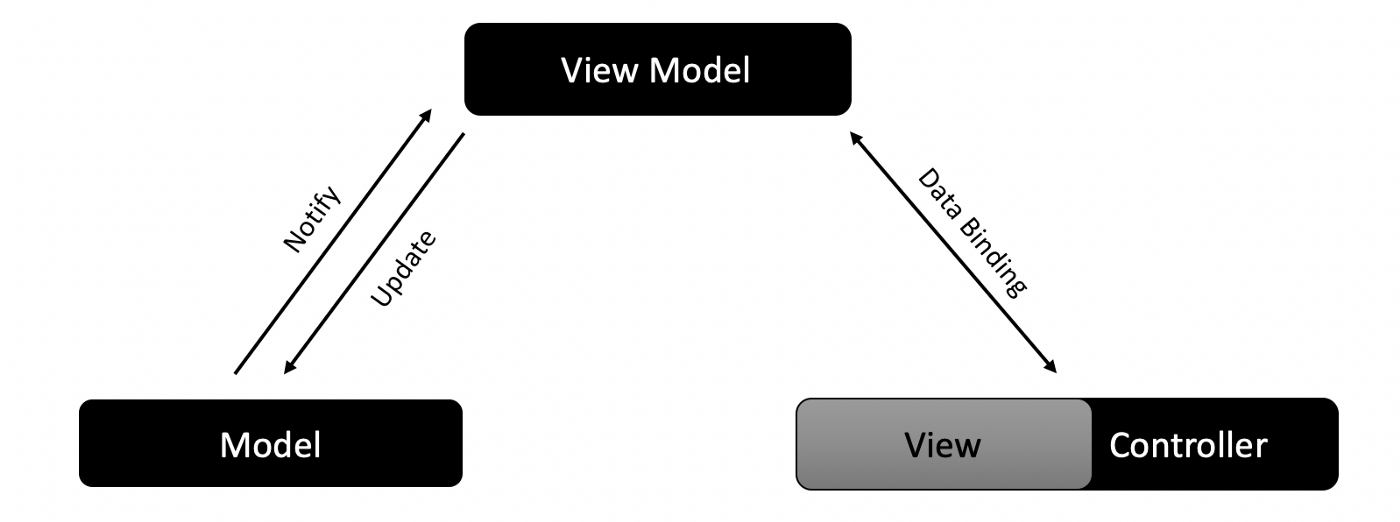
* There are still a tight coupling between View and Presenter.
* The Presenter is not reusable component, because it depends on a specific type, for example suppose we have register form with the same view as the update profile and the same logic, we will not able to reuse the Presenter because it depends on UpdateProfileViewController instance not RegisterViewController.
* The Presenter will not be fully testable because it depend on UIKit elements and UIKit elements can not be mocked.
* To solve this, let’s use Protocols

+ The idea here is to make the communication between Presenter and View done through Protocols in order to limits the Presenter to call some functions only in the ViewController not all function.

+ Another important thing that Protocols solves is the problem of reusability.

1. **MVVM (Model – View – ViewModel)**

The **MVVM** pattern is derived from **Model-View-Controller** pattern the same as the **MVP**, it also has the same components and relations as the **MVP**, but the **Presenter** Layer is replaced by the **ViewModel** Layer and the communication between the **View** and the **ViewModel** done through **Data Binding** approach



The Model and View are the same the MVP pattern let’s focus now on the **ViewModel**

What is the **View-Model**? and what are its **responsibilities**?

The **ViewModel** responsibilities are the same as the Presenter in MVP in addition to two points:

* The **ViewModel** should be hocked up with View via Data Binding paradigms, the most widely used **is Reactive Programming**, there are many frameworks which implements Reactive programming like RxSwift, ReactiveCocoa and Bond
* The **ViewModel** should represent the **View’s current state** at any time, and this means that, the ViewModel is a model that represents the View literally(for example if we have login screen that has two UITextfields for username and password. Then, the ViewModel should have two String properties which represents the username Textfield and password TextField) and this is very easy when you implement it using any reactive framework or using any data binding method

Ref: <https://medium.com/@dev.omartarek/mvp-vs-mvvm-in-ios-using-swift-337884d4fc6f>

<https://medium.com/flawless-app-stories/mvvm-in-ios-swift-aa1448a66fb4>

<https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52>