# Architecture & Design Pattern
# Research Paper

Throughout the scope of programming in general there are many different architectures out there. For Swift, the most commonplace are the MVC, MVVM, MVP, VIPER and even the FLUX architectures. There are also many different design patterns as well.

As we have iterated before in previous papers, the **MVC a**rchitecture is the Model View Controller. This is the most simplistic architecture of the bunch and can be very easy to understand and implement. The MVC in a nutshell has a view that takes user input and presents UIlayouts which will interact with the controller and if the controller ,if needed, will notify the model who then passes back data to the controller who then passes back the needed data to the view. Although it has a simplistic nature the MVC architecture has a running joke of being a "massive view controller". It lacks modularity and with more complex applications it can become overly complicated and hard to read. More complex applications will stress the controllers and tone down user experience. Because of MVC's shortcomings ,many have gone over to integrating the MVVM architecture.

**MVVM,** Model View View-Model, in itself is an architecture that utilizes yet another component called the View-Model, that interacts with the model and passes back data to the view. The new component View-Model basically takes the data from the model and passes it back to the view. The View-Model stands in between the model and view and imitates UIComponents by binding values and linking logical data to the UI. A disadvantage to MVVM in IOS is poor reusability. In MVVM, given how much the view model is doing, it may not be able to format a delegate or data source into a table view. This causes another view model needing to be created in order to format the needed data. Another con is testability due to the fact

that if you wanted to test a specific method it may need to run through a series of other methods first to retrieve the data needed in the method in question for testing. Yet another potential con is that if data binding isn't being utilized , you may have to write more code to achieve a specific end goal that the view controller can achieve in a shorter amount of code. This goes back to the original point of MVVM and does the opposite by not reducing the stress on view controllers. This in turn leads to an alternative to better ease stress distribution in the form of the MVP architecture.

The **MVP** architecture, Model, View, and Presenter, introduces the presenter component. This architecture divides objects in the three components in MVP. The view is now the view and view controller, the model contains the application data, networking calls, parser, extensions, and manager while the presenter is in charge of business logic, user action definitions, and UI updates by using delegates. Since the presenter is not reliant on UIKit it makes it testable.A con of MVP is maintainability due to the fact that the view model and view are one. Another con is that the presenter may not accurately represent the view.

The **VIPER** architecture,View Interactor Presenter Entity Router, which separates code by single responsibility. It , just like MVVM, tries to reduce the stress of view controllers. The view is the user interface, the interactor mediates between the presenter and data, the presenter directs data between the view and interactor, the presenter represents app data, and the router is in charge of navigation between screens.For cons of Viper im going to break character here and share a funny thing i read up on it. I read that VIPER is "what happens when former enterprise Java programmers invade the iOS world "😂. Formally, cons for VIPER include redundant components such as the presenter due to the fact that it only proxies calls from the view to the interactor. Another con is that it is a complex architecture and unneeded in smaller applications.

A major con is that the application must have stable specification in order to implement VIPER due to the fact you will have to refactor a large amount of code to fix single issues.

One other architecture is the **FLUX** architecture. This architecture uses the four components: Action, dispatcher, store, and view. Action being the user interaction from the view, the dispatcher being a central hub that is observed by the store, the store that holds the business logic , and the view being the UI representation. Cons to this architecture can be the complexity and a case where there are many views that map to only a single store.

Now just as there are different architectures out there ,there are also many different design patterns as well. A very useful design pattern is the singleton design pattern. The **singleton** is a creational pattern that makes sure that only one instance of a particular class is instantiated and that the application only uses that instance. The best use of singletons would be in a case that you need to create one as a way to provide unified access to a point, resource, or service that's shared throughout the app such as a network manager making http requests or an audio channel that plays sound effects, Managing shared resources is the primary use.

The **Factory** design pattern is a protocol that returns new objects.This design pattern provides an interface for creating objects in a superclass. The factory design pattern is used mainly when you need to separate product creation logic. It's a creational design pattern that allows us to have objects use other instances of other objects without having to be initialized.Its also imperative that an interface is created in order to be implemented.

The **facade** design is a structural design pattern that creates an interface for complex class systems, framework, or library. It's the ideal structure for when complexity is a hurdle. So we have a factory that provides an interface and a facade that provides simplified data to the

interface. This is not an implementation but rather an interface. Its main job is to protect the client from the complexity of sub system components.

The **decorator** design pattern is a pattern that adds new behaviors to objects by putting them inside special wrapper objects. It's classified as more of a structural design pattern. It allows us to wrap objects an unlimited amount of times without modifying the objects code. Decorators at their core are just flexible alternatives to subclassing.

The **Adapter** design pattern is a structural design pattern that allows objects that have incompatible interfaces to work with one another. This is usually used when transferring legacy code into modern code. It's basically a useful translator. The Adapter is a wrapper between two objects and transforms the interface interpretation from one object to the formatted recognizable interface by the second object.

The **Bridge** design pattern is a structural design pattern that divides business logic into separate class hierarchies. The Bridge pattern is typically used when dealing with cross-platform apps, supporting multiple types of database servers or working with several API providers of a certain kind. Sometimes the adapter and bridge design patterns can work together hand and hand to properly translate suitable legacy code.

The **Composite** design pattern is a structural pattern that composes objects in a tree structure and allows it to function as a singular object. It’s primarily used for composing classes and objects into a larger system. The client can think that it is interacting with one object but really it is interacting with a collection of objects behind the scenes.

All in all, there is no perfect architecture out there, each has their fair share of pros and cons. The decision to implement one of these architectures is widely based on the specifications and

requirements needed from project to project. The same can be said on design patterns, they can all be useful or applicable given the application.