Architecting Better iOS Applications

...

Abhishek Sheth
Appbirds Technologies

Agenda

Why do we need Architecture for developing apps

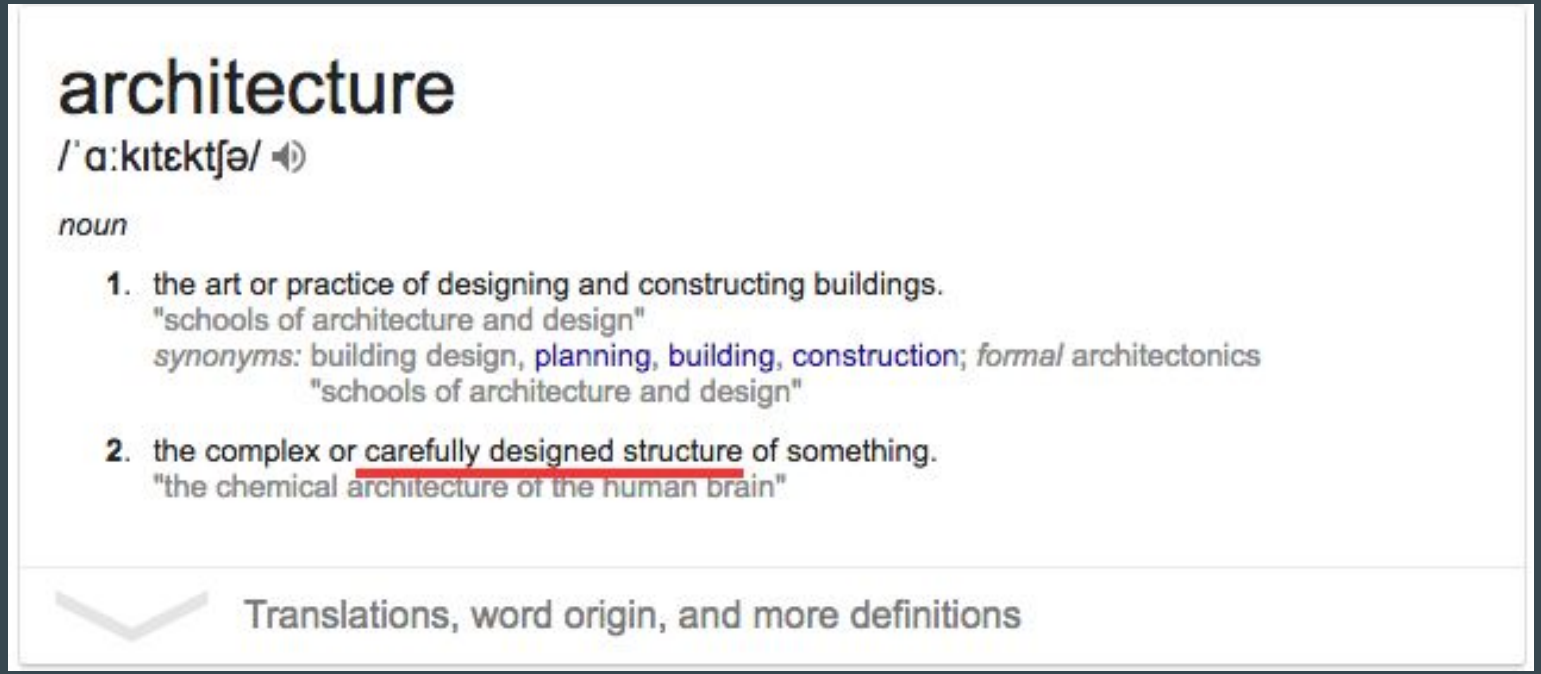
Most preferred architecture for developing iOS Apps (MVC)

Massive View Controllers

Refactoring Demo

Looking Beyond MVC (MVVM, VIPER)

Meaning of Architecture



architecture

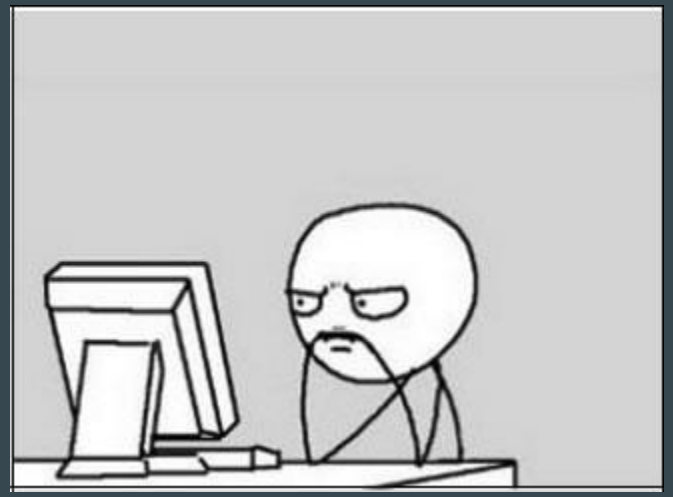
/ˈɑːkɪtɛktʃə/

noun

1. the art or practice of designing and constructing buildings.
   "schools of architecture and design"
   synonyms: building design, planning, building, construction; *formal* architectonics
   "schools of architecture and design"

2. the complex or carefully designed structure of something.
   "the chemical architecture of the human brain"

Translations, word origin, and more definitions



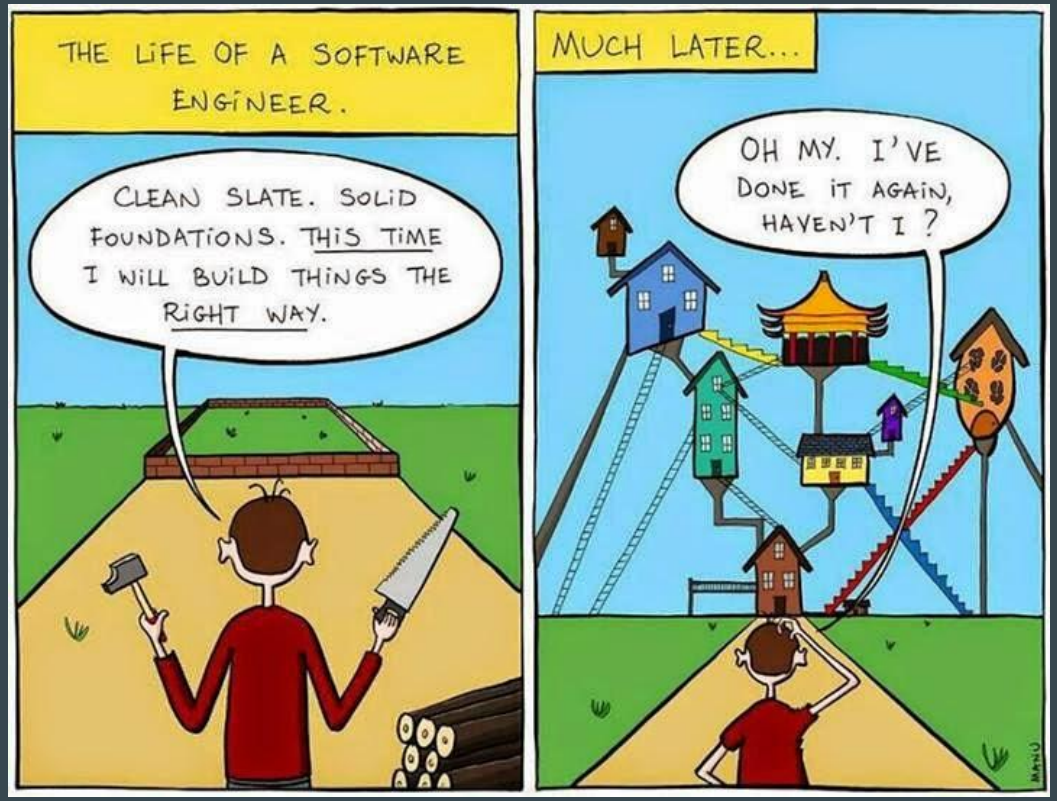
Why should I care about architecture?

You don’t need to worry about architecture if.....

If you are building the apps which has almost zero functionalities

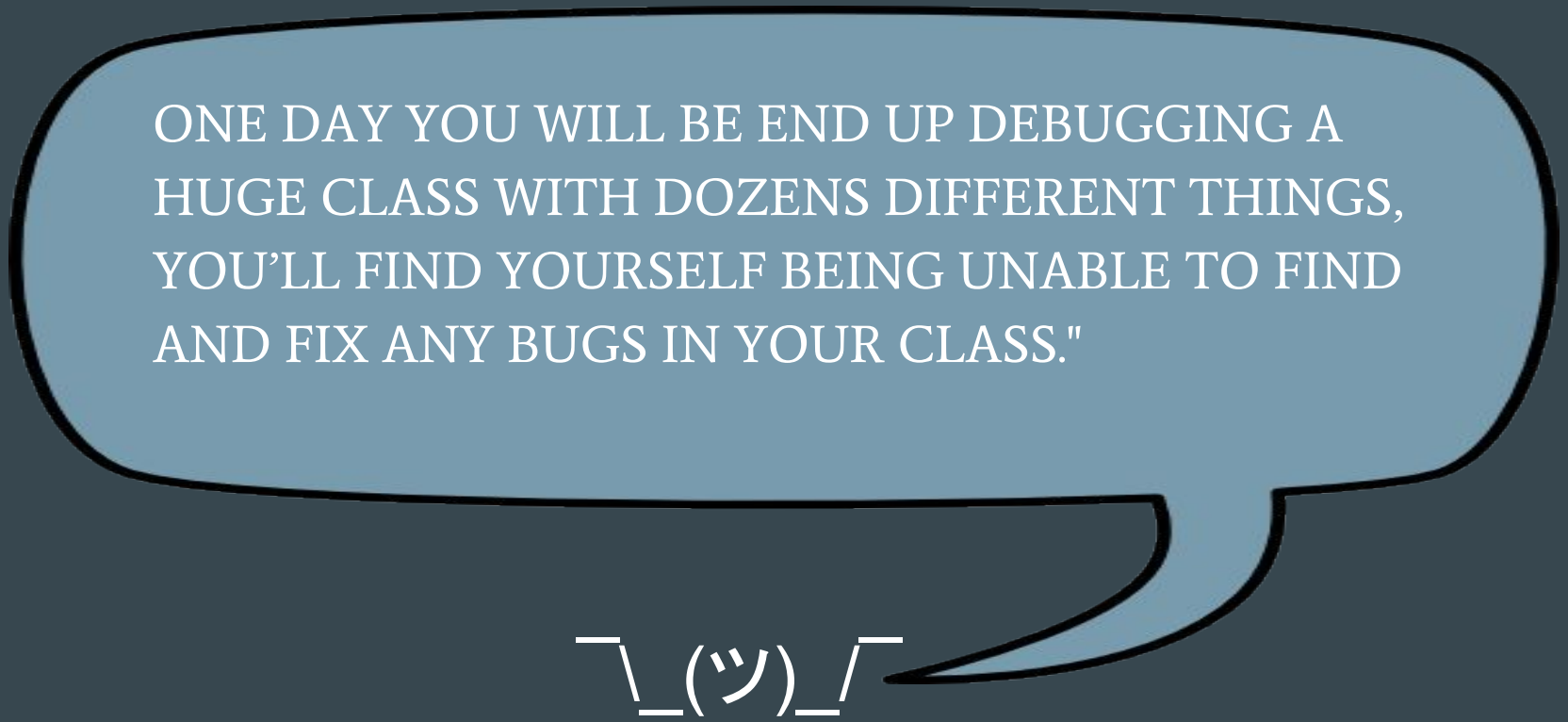
- Apps that just change background color
- Application with just one webview

But unfortunately such applications are no longer approved by Apple!!



Beginning of any new project

If you don’t care about architecture then..



Why? Because.....

We have very extra extra large (XXXXXXXXL) UIViewController subclasses where

Data is directly stored in the view controllers

It handles all the network communication

It handles entire view lifecycles

It takes care of navigation logic

Models and Views are just plain subclasses

No tests written at all

So the end product would be like

What can we achieve with a good architecture?

Distribution of responsibility among all components

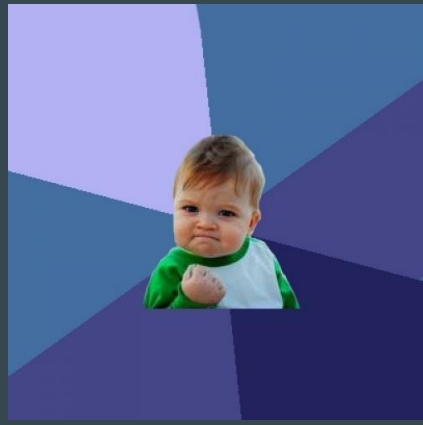
Easily readable and understandable code for other developers

Less bugs, and faster bug fixes

Code can easily change when requirements change

Better Code reusability between same project and multiple projects

Clean and testable code that you can take pride of



Architecture patterns for iOS Applications

~~Model~~ Massive View Controller (Most preferred)

Model View View-Model (MVVM)

VIPER (View, Interactor, Presenter, Entity Router)

Unidirectional Data Flow (ReSwift)

Riblets (Discovered by Uber)

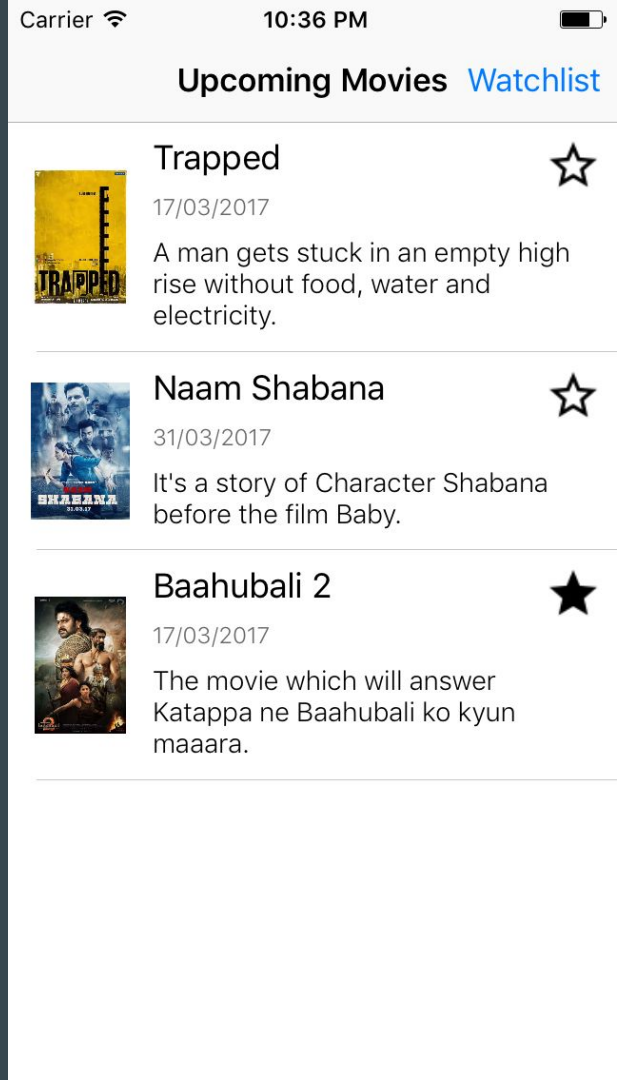
Example App

Fetches list of upcoming movies from servers

Show them as a list

Add/Remove a movie from watchlist

Watchlist screen which lists movies in watchlist



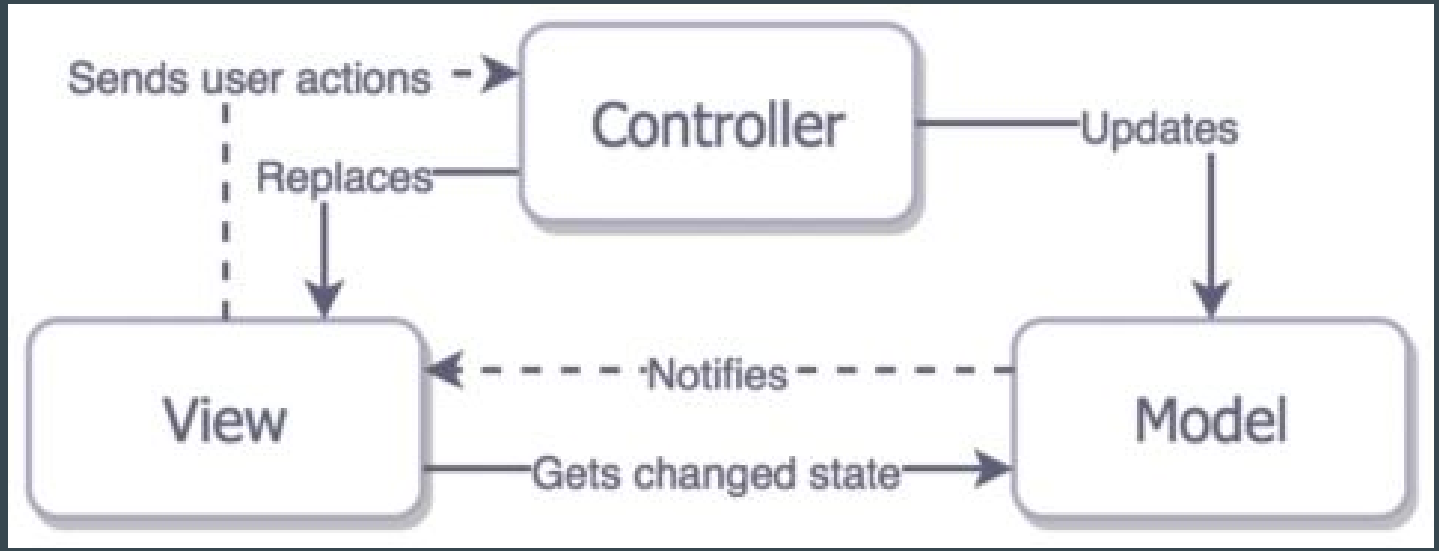
MVC - Model View Controller

Model - Which can be mapped with any business entity or data, and methods to access them (i.e Movie)

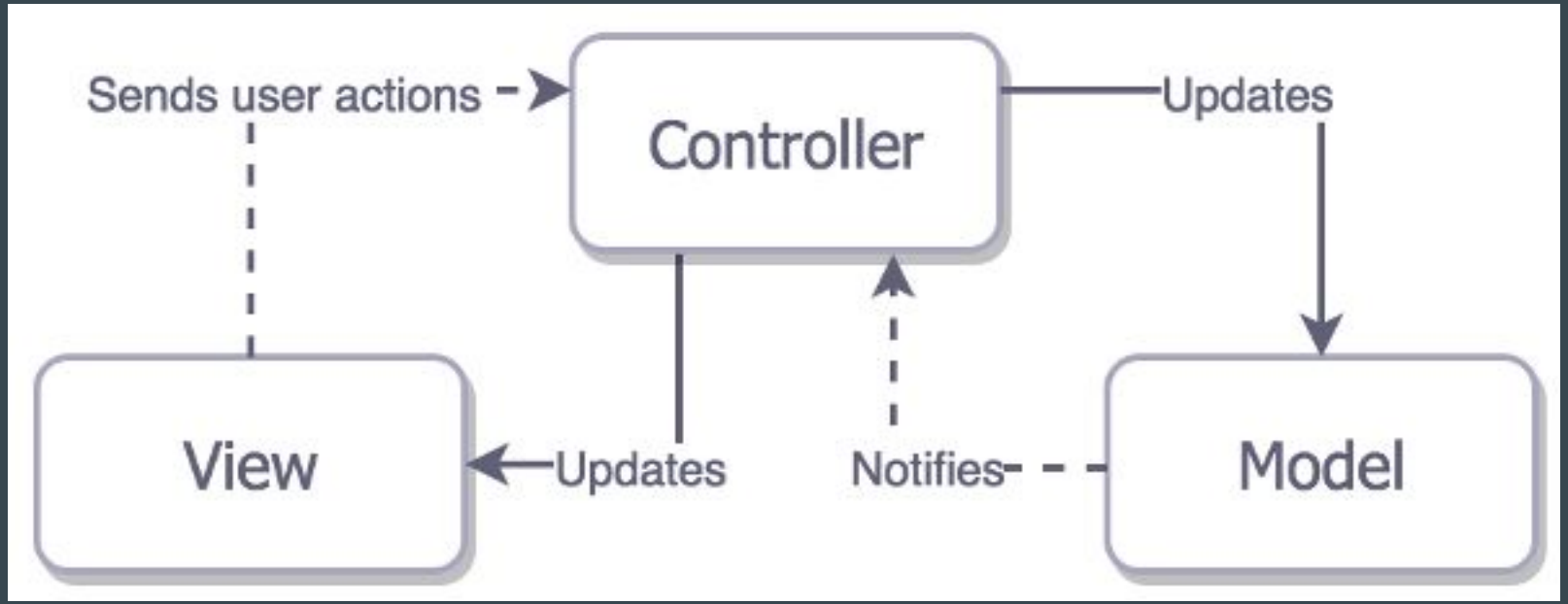
View - User Interfaces which are directly visible to the user, User can see and interact with views (i.e MovieListTableViewCell)

Controller - That connects view and model, takes input from the view, updates model according to it (i.e MovieListViewController)

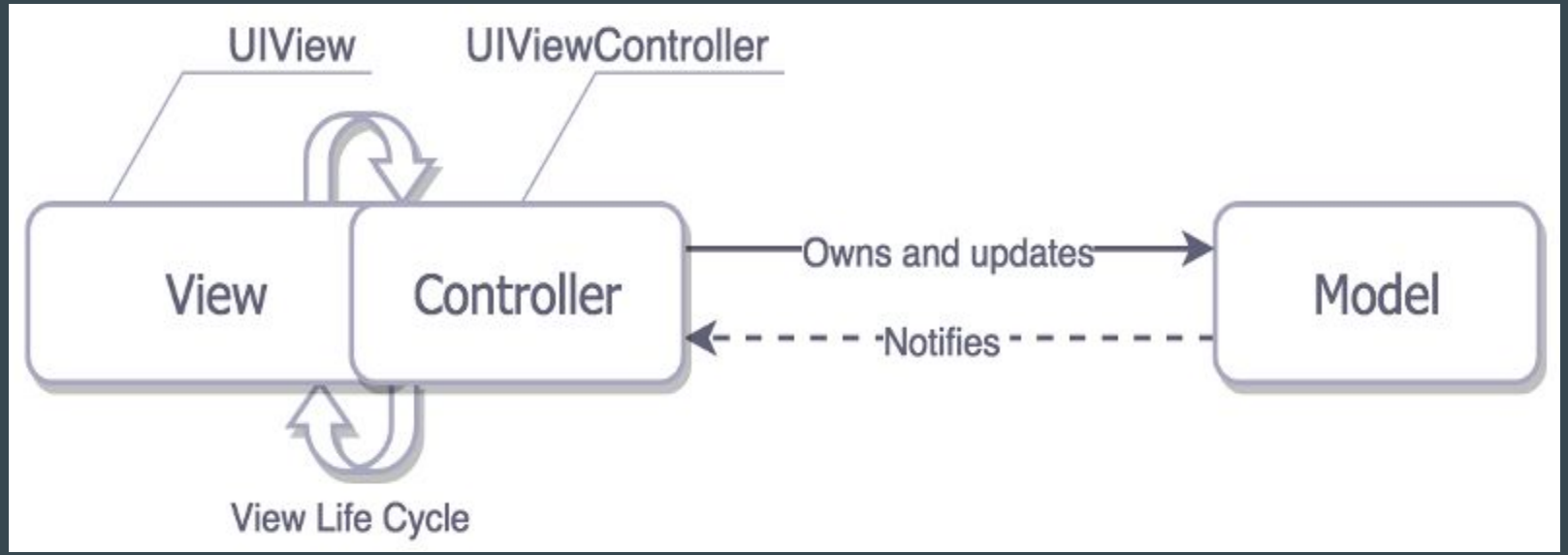
Classic MVC (How MVC was invented to be)



Model View Controller (How Apple thinks we are using it)



Massive View Controllers (How we actually use it)



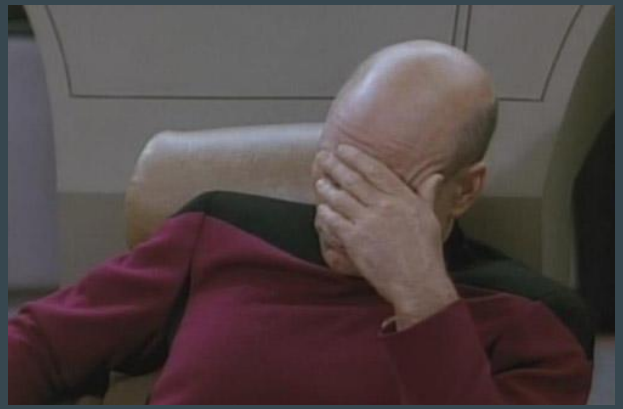
Massive View Controllers

View controller has code of more than 500 lines

View controller handles everything like network requests, tableview datasource delegate methods, view layout, and UI interaction

It grows bigger and bigger with changes

Most of the code is not reusable



Massive View Controllers Demo

How to avoid Massive View Controllers

DRY (Don’t repeat Yourself)

Move view manipulation code to view layer (i.e configure(with movie : Movie))

Move Json parsing code to Models (i.e init(with dictionary:[String:Any]))

Create Base classes and move repeated code there (i.e BaseTableViewController)

Use Extensions, Protocols

Refactoring Massive View Controllers With Base Classes & Extension Demo

Services & Managers

A service is an object that encapsulates the logic that related to one task or group of similar tasks.

Service can decide whether to fetch data from local storage or make a network call to fetch the data

Managers are mostly singleton classes which take cares of particular tasks which are device dependents

Example : Network manager can have all the code related to networking

Location manager takes care of location related functionality

Refactoring Massive View Controllers With Services & Managers Demo

Separate Datasource

Generally we put the tableview handling in the view controller it self which leads to Massive View Controller and that logic can not reuse among other class

Also it impacts testability of the class

Better to create separate datasource classes, which can be reusable

Refactoring Massive View Controllers By Separating Datasource Demo

Flow Coordinators

General navigation flow in iOS Application would be like

```
1  extension MasterVC: UITableViewController {
2          func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
3
4                  let detailData = getDetailData(indexPath: indexPath)
5
6                  let  detailVC = DetailVC()
7                  detailVC.detailData = detailData
8
9                  self.navigationController.pushViewController(detailVC, animated: true)
10         }
11 }
```

So what' s wrong with it

The Navigation Controller is a parent of MasterVC, and yet its child commands it. Ideally child controller should not even have idea about their parent controllers

It’s perfectly fine for a child ViewController to delegate some work, even if it delegates them to the parent. However, when the child VC has a reference to the parent ViewController and directly calls its methods, the whole thing becomes problematic.

It increases excessive coupling and gives UIViewController much responsibilities

Example: App startup

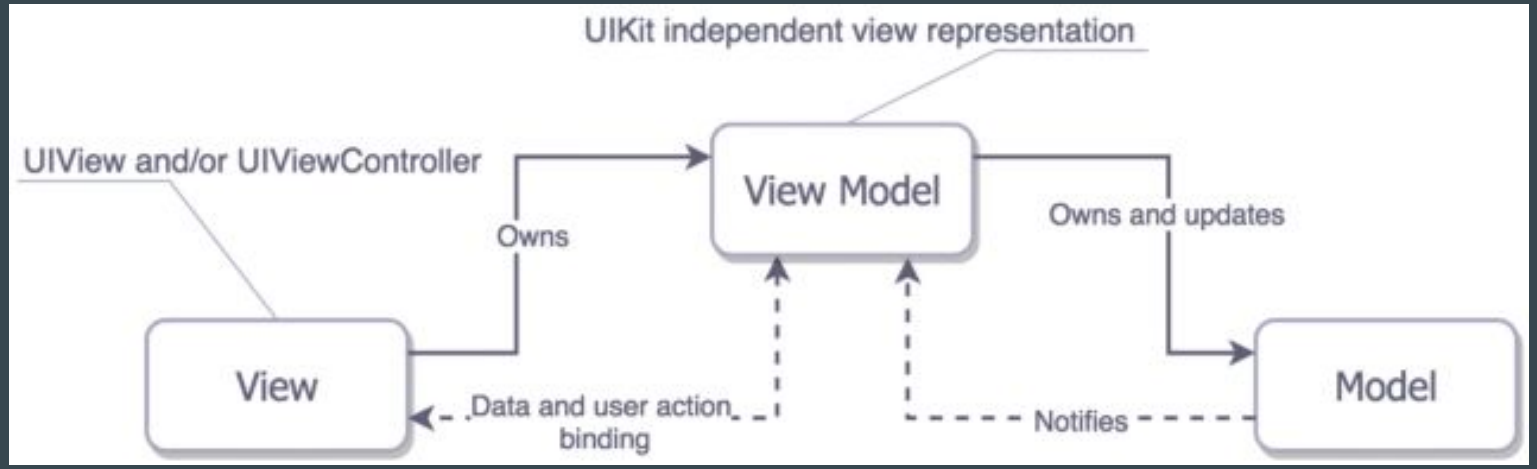
Example : App startup flow

```
class RootCoordinator {
        func start() {
                if userLoggedIn() {
                        showHomeCoordinator()
                } else {
                        showAuthenticationCoordinator()
                }
        }

        func showHomeCoordinator() {
                let navigationController = UINavigationController()
                let homeCoordinator = HomeCoordinator(navigationController: navigationCon

                unowned(unsafe) let coordinator_ = homeCoordinator
                homeCoordinator.onEnding = { [weak self] in
                        self?.removeChildCoordinator(childCoordinator: coordinator_)
                }

                addChildCoordinator(homeCoordinator)
                homeCoordinator.start()
```

```swift
func application(_ application: UIApplication, didFinishLaunchingWithOptions launch

        ...
        let rootVC = UIViewController()
        window = UIWindow(frame: UIScreen.mainScreen().bounds)
        window!.rootViewController = rootVC
        window!.makeKeyAndVisible()

        let window = UIWindow(frame: UIScreen.main.bounds)

        rootCoordinator = RootCoordinator(rootViewController: rootVC)
        rootCoordinator.start()

        return true
}
```

Looking Beyond MVC (If you are feeling adventurous ;))

MVVM


UIKit independent view representation
UIView and/or UIViewController
View Model
Owns
Owns and updates
View
Data and user action binding
Notifies
Model

Advantages

View Controller will become a lot less bloated. If we want custom formatting, manipulation with the Model data, we can easily do it in the View Model

Lower coupling between Model and View Controller

You can test ViewModel separately

Difficulties:

MVVM binding better when used with functional or reactive programming, which has steep learning curve

VIPER

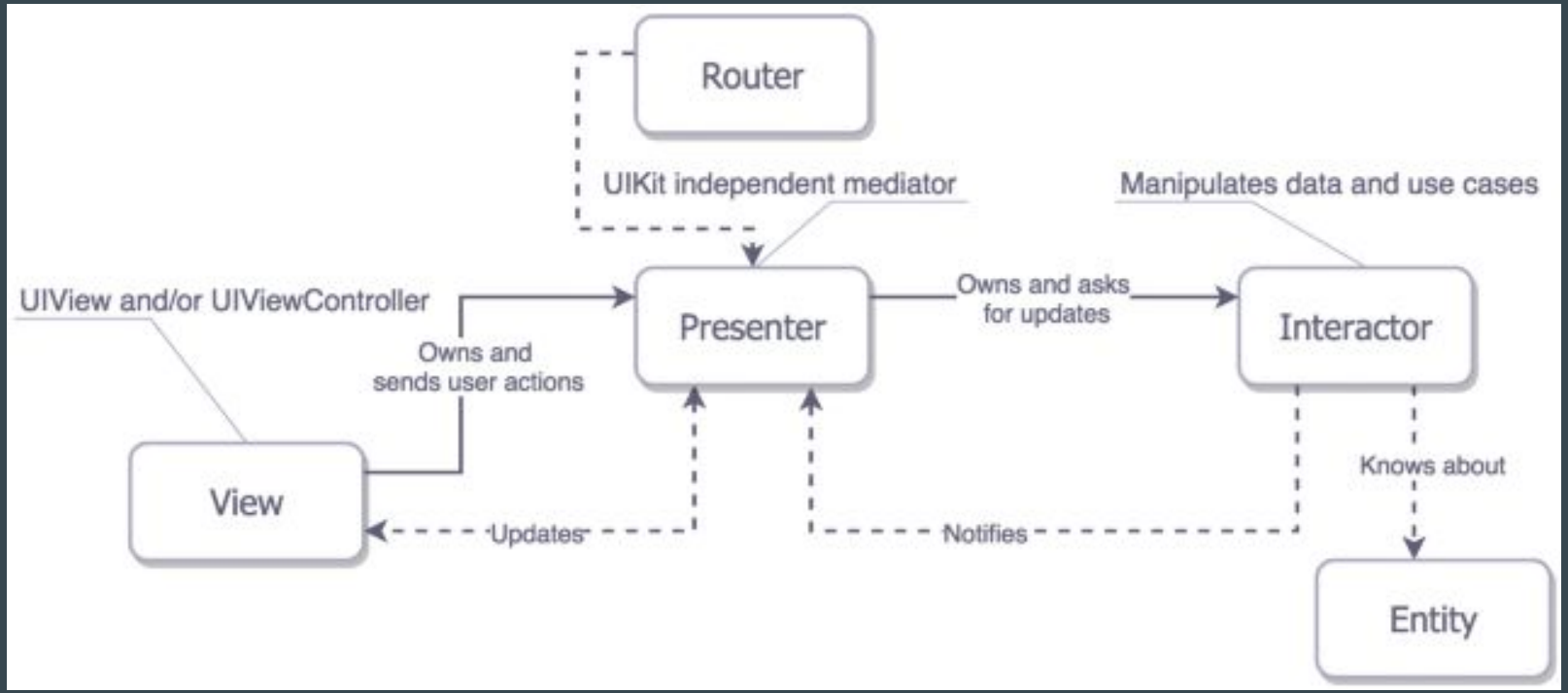
View: displays what it is told to by the Presenter and relays user input back to the Presenter.

Interactor: contains the business logic as specified by a use case.

Presenter: contains view logic for preparing content for display (as received from the Interactor) and for reacting to user inputs (by requesting new data from the Interactor).

Entity: contains basic model objects used by the Interactor.

Router: contains navigation logic for describing which screens are shown in which order.


Router
UIKit independent mediator
Manipulates data and use cases
UIView and/or UIViewController
Presenter
Owns and asks
for updates
Interactor
Owns and
sends user actions
View
Updates
Notifies
Knows about
Entity

VIPER Modules Demo

Advantages

You can easily interchange components (i.e API to Local Storage)

View Controllers becomes very small, Good bye to Massive View Controllers

Multiple people can work on same module

Each component can be tested separately

Easy to fix bugs

Before you jump into the VIPER

VIPER architecture should be used for applications whose requirements are very well defined at the beginning

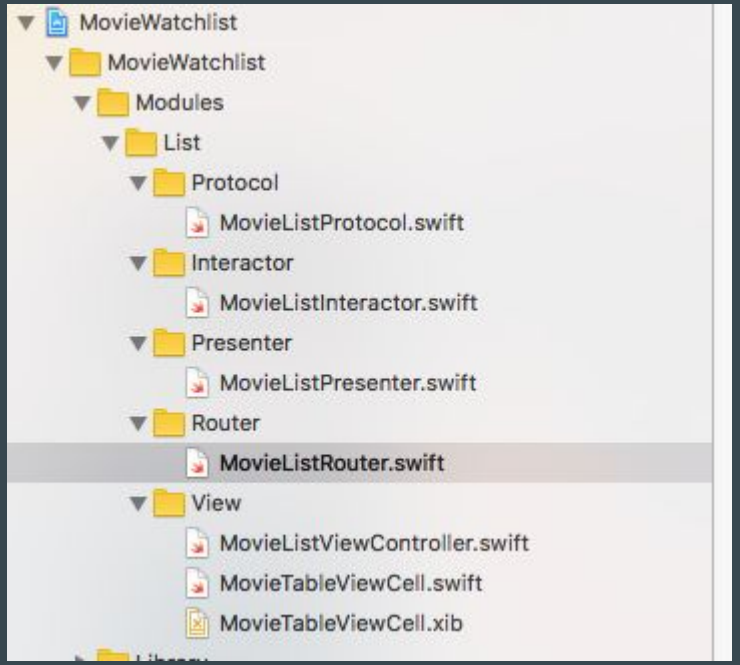
Everyone in the team should know it in details

It can be over-engineering for small-mid projects

Easily affect your deadlines

Tough for new developers to understand

Folder Structure in VIPER



Conclusion

Select architecture which suits your requirement, don't select architecture just because it looks cool

It's perfectly alright to mix and match architecture.

Even MVC is fine if it's not leading to Massive View Controllers

All the best for your next project :)

Thank You :)