**Popcorn**

v 1.0

Developed by Hashem Abounajmi

**Popcorn** is written with an architecture first introduced by *“Robert Cecil Martin”* in a book named: *“Clean Architecture”.*  now applied to iOS using swift named: *“Clean swift”.* the benefits are:

• Find and fix bugs faster and easier.

• Change existing behaviors with confidence.

• Add new features easily.

• Write shorter methods with single responsibility.

• Decouple class dependencies with established boundaries.

• Extract business logic from view controllers into interactors.

• Build reusable components with workers and service objects.

• Write factored code from the start.

• Write fast and maintainable unit tests.

• Have confidence in your tests to catch regression.

• Apply what you learn to new and existing projects of any size

# Organizing Code

We’ll create a new single view application. then create a nested subgroup **Scenes -> SearchMovies**. When we implement show movie details use case in the future, we’ll create a new sub-group **Scenes -> ShowMovieDetails**

In a typical Xcode project, it is common to see files organized into model, view, and controller groups. But every iOS developer knows MVC. It doesn’t tell you anything specific about the project. As Uncle Bob pointed out, group and file names should reveal your intentions for the use cases. It should not reflect the underlying framework structure. So we’ll organize each use case under a new group nested within Scenes.

Inside the SearchMovies group, you can expect all files have something to do with searching a movie. If you see a new ViewMovieDetails group created by another developer, you already know what to expect.

This organization tells you far more than the model, view, and controller groups you are used to see. Over time, you’ll accumulate 15 models, 27 view controllers, and 17 views. What do they do? You simply don’t know before you inspect each file.

# The VIP Cycle

The view controller, interactor, and presenter are the three main components of Clean Swift. They act as input and output to one another as shown in the following diagram.

The view controller’s output connects to the interactor’s input. The interactor’s output connects to the presenter’s input. The presenter’s output connects to the view controller’s input. We’ll create special objects to pass data through the boundaries between the components. This allows us to decouple the underlying data models from the components. These special objects consists of only primitive types such as Int, Double, and String. We can create structs, classes, or enums to represent the data but there should only be primitive types inside these containing entities.

This is important because when the business rules change that result in changes in the underlying data models. We don’t need to update all over the codebase. The components act as plugins in Clean Swift. That means we can swap in different components provided they conform to the input and output protocols. The app still works as intended.

A typical scenario goes like. The user taps a button in the app’s user interface. The tap gesture comes in through the IBActions in the view controller. The view controller constructs a request object and sends it to the interactor. The interactor takes the request object and performs some work. It then puts the results in a response object and sends it to the presenter. The presenter takes the response object and formats the results. It then puts the formatted result in a view model object and sends it back to the view controller. Finally, the view controller displays the results to the user

# 1. ViewController

First we setup the scene by calling setup() method. to set up the VIP chain. after user entered a search term and hit search button, we call search(withSearchTerm), we create a searchRequest object and invoke searchMovies() on the interactor. That’s it. We ask the output to perform our business logic. The view controller doesn’t and shouldn’t care who and how it is done.

# 2. Interactor

The interactor contains your app’s business logic. The user taps and swipes in your UI in order to interact with your app. The view controller collects the user inputs from the UI and passes it to the interactor. It then retrieves some models and asks some workers to do the work.

when user hit search button, interactor receives searchMovies(request:) call. The argument of this method is the request object of type SearchMovies request. The interactor peeks inside this request object to retrieve any necessary data to do its job which is now the search term. here Interactor asks the worker object to search for the movies that matches the search term by calling searchMovies(withSearchTerm, page). It then constructs a response object and invokes presentSearchedMovies() on the output which is the presenter

# 3. Worker

The SearchMoviesWorker provides an interface and implementation of the work it can do to the interactor. it search for the movies for the target page, then if search was successful, it stores the search term, later used to display recent searches, then return the founded movies

# 4. Presenter

After the interactor produces some results, it passes the response to the presenter. The presenter then marshal the response into view models suitable for display. It then passes the view models back to the view controller for display to the user.

Since the output of SearchMoviesInteractor is connected to the input of SearchMoviesPresenter, the presentSearchedMovies(response:) method will be called after the interactor finishes doing its work. It simply constructs the view model object and invokes displaySearchedMovies(viewModel:) on the output.

displaySearchedMovies(viewModel:) is the last step in the VIP cycle. It takes any data in the view model object and displays it to the user. Here it receives raw movies objects from interactor then it maps to DisplayMovie structs then passes them to the viewController.

# 5. Router

When the user taps the next button to navigate to the next scene in the storyboard, a segue is trigged and a new view controller is presented. A router extracts this navigation logic out of the view controller. It is also the best place to pass any data to the next scene. As a result, the view controller is left with just the task of controlling views.

# Models

In order to completely decouple the Clean Swift components, we need to define data models to pass through the boundaries between them, instead of just using raw data models. There are 3 primary types of models:

* **Request** – The view controller constructs a request model and passes it to the interactor. A request model contains mostly user inputs, such as text entered in text fields and values chosen in pickers.
* **Response** – After the interactor finishes doing work for a request, it encapsulates the results in a response model and then passes it to the presenter.
* **View Model** – After the presenter receives the response from the interactor, it formats the results into primitive data types such as String and Int, and stuff them in the view model. It then passes the view model back to the view controller for display.