Clean Swift

Link to explanation of below params.

Param	Metrics	Value
Boiler plate	Low / Medium / High	**Medium** (architecture expects a specific project structure and complex set of files when creating a new ViewController to maintain a protocol-oriented VIP cycle with routing; but its highly adjustable regarding other parts like database, analytics, services as it can be put in multipurpose workers)
Learn ing Curve	Easy / Medium / Hard	**Medium** (easier than VIPER, harder than MVVM)
Matur ity	From what year	**2016**
Usag e in NG	Low / Medium / High	**Low**
Code bindin g level	Low / Medium / High	**Low** (Fully native, no test or implementation frameworks required)

Quick overview 👀 :

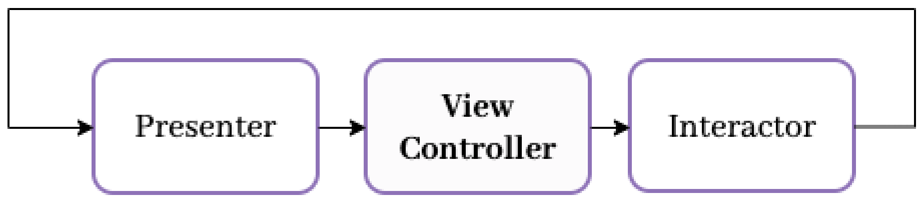
Clean Swift (VIP) was first introduced by Raymond Law on his website clean-swift.com. The idea behind it was to tackle the Massive view controller problem while following the main ideas found in Uncle Bob's Clean Architecture.

When implementing a Clean Swift project your code will be structured around each of your application screens or segments of screens also known as “scenes”.

In theory, each scene is a structure with around 6 components:

- View Controller
- Interactor
- Presenter
- Worker
- Models
- Router

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 output of the view controller is connected to the input of the interactor. The output of the interactor is connected to the input of the presenter. The output of the presenter is connected to the input of the view controller. This means the flow of control is always **unidirectional**.”*

Example:

Imagine a screen with a login button. It's a Scene that defines a structure with a VIP cycle of View Controller, Interactor, and Presenter. When the user taps the button, View Controller calls the interactor. Interactor uses the business logic inside to prepare an output (with the use of workers). Then propagates the result to the presenter. The presenter calls the VC's method to call the router to display a new scene.

View Controller

- Defines a scene and contains a view or views
- Keeps instances of the interactor and router
- Passes the actions from views to the interactor (output) and takes the presenter actions as input

Interactor

- Contains Scene's business logic
- Keeps reference to the presenter
- Runs actions on workers based on input (from the View Controller) triggers and passes the output to the presenter
- Interactor should never import UIKit

Worker

- An abstraction that handles different underhood operations like fetch the user from Core Data, download the profile photo, allows users to like and follow, etc.
- Should follow the Single Responsibility principle (interactor may contain many workers with different responsibilities)

Presenter

- Keeps a weak reference to the view controller that is an output of the presenter
- After the interactor produces some results, it passes the response to the presenter, presenter then marshal the response into view models suitable for display and then passes the view models back to the view controller for a display to the user

Router

- extracts this navigation logic out of the view controller
- keeps a weak reference to the source (View Controller)

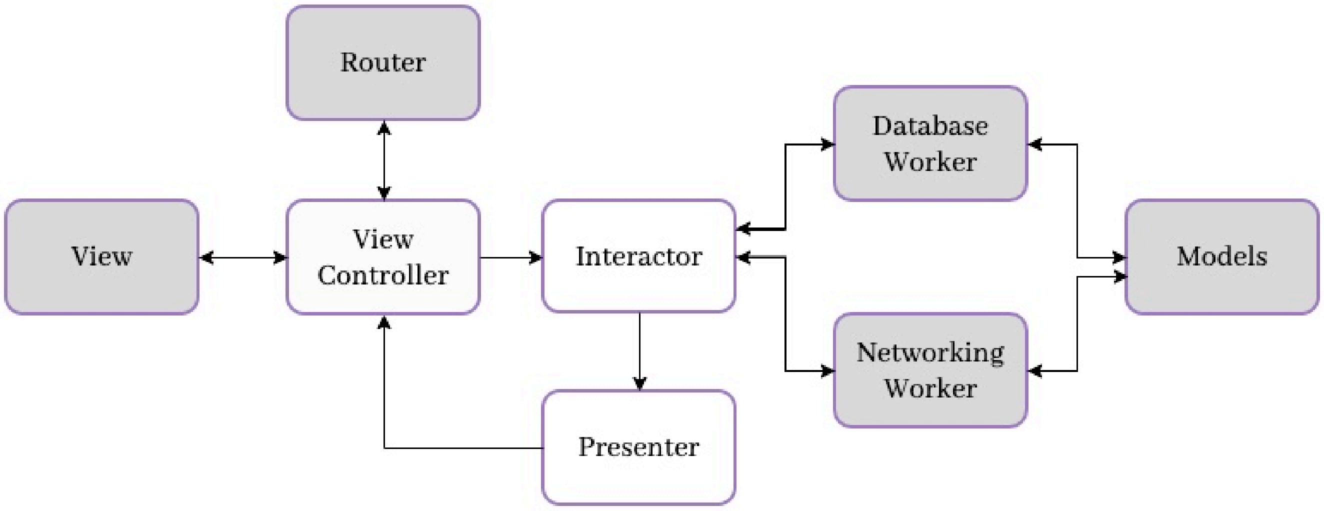
Configurator

- takes the responsibility of configuring the VIP cycle by encapsulating the creation of all instances and assigning them where needed

Model

- decoupled data abstractions

Full Schema:



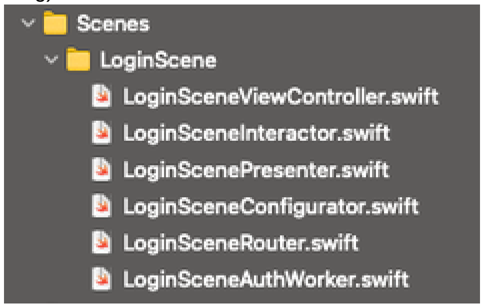
Other:

To **avoid memory leaks** always pass the view controller to the router and presenter as a weak reference.

I'm not the biggest fan of the official implementation guide. I encourage you to check the sample project of CleanStore or Sample Project using Clean Swift: NYTimes instead. Or just follow the examples above (examples are from my sample project on GitHub).

Key Rules :ubs-keys: :

- Keep the file structure (follow the Scene naming)



```
Scenes
  LoginScene
    LoginSceneViewController.swift
    LoginSceneInteractor.swift
    LoginScenePresenter.swift
    LoginSceneConfigurator.swift
    LoginSceneRouter.swift
    LoginSceneAuthWorker.swift
```

- Use VIP cycle and input/output protocols:
  - The view controller accepts a user event, constructs a request object, sends it to the interactor
  - The interactor does some work with the request, constructs a response object, and sends it to the presenter
  - The presenter formats the data in the response constructs a view model object and sends it to the view controller
  - The view controller displays the results contained in the view model to the user

Recommendations ⭐ :

- projects where unit testing is expected
- long-term and big projects
- projects with a generous amount of logic
- projects you want to reuse in the future
- when MVVM, MVP, MVC is not enough or you just hate VIPER but there is a need to introduce a sophisticated architecture
- native and imperative projects

Strengths 💪 :

- Easy to maintain and fix bugs

- Enforces modularity to write shorter methods with a single responsibility
- Nice for decoupling class dependencies with established boundaries
- Extracts business logic from view controllers into interactors
- Nice to build reusable components with workers and service objects
- Encourages to write factored code from the start with fast and maintainable unit tests
- Applies to existing projects of any size
- Modular: Interfaces may be easy to change without changing the rest of the system due to using a protocol conformance business logic
- Independent from the database

Disadvantages and traps 👎:

- Many protocols with complicated naming and responsibilities, may be confusing at first where is the protocol defined
- High app size usage due to many protocols
- Despite the fact, there is an official website of Clean Swift architecture it changes and implementations may differ between projects
- It's hard to maintain the separation between VC and Presenter, sometimes presenter just calls the view methods instead of preparing the UI so it seems useless and just creates boilerplate

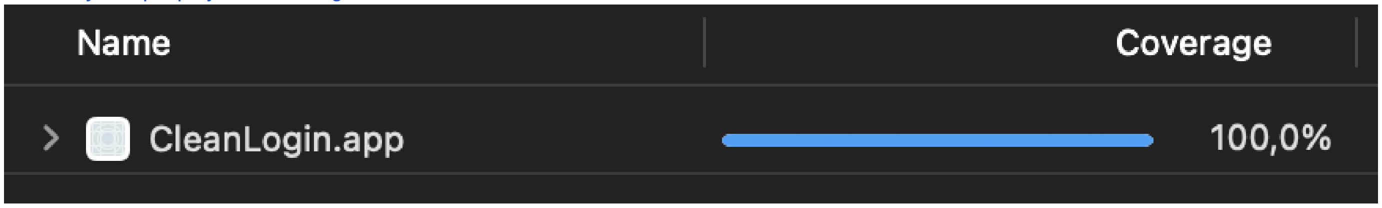
Learning materials:

- Clean Swift VIP with Example
- My sample project on GitHub: CleanLogin
- Sample Project using Clean Swift: NYTimes
- Official implementation guide
- Another sample project: CleanStore
- What is clean swift(VIP), a brief overview
- Clean Swift Architecture: Is It Really Good for Your App?
- Introducing Clean Swift Architecture (VIP)
- The Clean Swift architecture explained

Summary 🧐:

Clean Swift is not the easiest to maintain but possibly the best to avoid coupling and be able to keep the high code coverage score.

VIP cycle enforces a strict naming convention and class responsibilities with a medium amount of boilerplate code.
Would you like to see a simple example of Clean Swift architecture?
Check my sample project 'CleanLogin' on GitHub.



Name	Coverage
CleanLogin.app	100,0%