Luong Nguyen Khoi Nguyen

Application of Protocol-Oriented MVVM Architecture in iOS Development

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

25 April 2017

Author(s) Title Number of Pages Date	Luong Nguyen Khoi Nguyen Application of Protocol-Oriented MVVM Architecture in iOS Development 42 pages + 1 appendices 25 April 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Smart Systems
Instructor(s)	Patrick Ausderau, Senior Lecturer Vu Nguyen, Instructor

The mobile application industry is fast paced. Requirements change, additions of new features occur on a daily basis and demand frequent code structure adjustment. Thus, a flexible and maintainable software architecture is often a key factor for an application's success. The major objective of this thesis is to propose a practical use case of Protocol Oriented Model View View Model, an architecture inspired by the Protocol Oriented Programming paradigm.

This thesis explains the architecture concepts by analyzing the process of constructing a Protocol Oriented Model View View Model-oriented Swift application. Moreover, it indicates the uses of software development's best practices in specific situations. Drawbacks of modern architecture like Model View Controller are also addressed.

The main product of this thesis is a fully functional iOS application written in Swift. Also, for learning purposes, parts of the application's code base are extracted, modified and can be freely used under the MIT license.

In conclusion, the thesis exhibits how a well-design architecture could affect the overall quality of an application, especially in terms of maintainability. It also encourages a best-practice minded approach in software development and further studies toward the practical use of the Protocol Oriented Model View View Model architecture.

Keywords	Model View View Model, Protocol Oriented Programming, Swift, Architecture, Trait, iPhone Operating System

Contents

Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DRY	Don’t repeat yourself
HTTP	Hypertext Transfer Protocol
iOS	iPhone Operating System
KVO	Key-Value Observing
MVC	Model View Controller
MVP	Model View Presenter
MVVM	Model View View Model
OOP	Object Oriented Programming
PHP	Hypertext Preprocessor
POP	Protocol Oriented Programming
POP MVVM	Protocol Oriented Model View View Model
VIPER	View Interactor Presenter Entity Router
WWDC	Worldwide Developers Conference

Glossary

Apple App Store	A digital distribution platform provided by Apple. It allows iPhone Operating System (iOS) users to browse and download applications developed with the development kit provided by Apple
business logic	Also known as domain logic, is a subset of the application logic whose main responsibility is to determine suitable operations on the data life cycle (creation, update, deletion, etc.) based on a real-world use case
callback	A function that is passed to a parameter list of another function and is triggered after some kind of event
FIRST	Abbreviation for: First, Isolated, Repeated, Self-validating and Timely, are the 5 basic principles for composing clean software testing proposed by Robert C. Martin [1, p. 132]
interface	In Object Oriented Programming (OOP), interface is a set of descriptions of actions that an object can do [2]
iOS	iPhone Operating System: a mobile operating system created and developed by Apple Inc. exclusively for its hardware

JSON	Stands for JavaScript Object Notation, is lightweight data-interchange format. It is human readable, easy to parse or generate and language independent text format [3]
lifecycle	The series of changes in state of a UIViewController. [4]
MIT	A permissive software license, originally developed at the Massachusetts Institute of Technology [5, p. 84]
Node.js	A platform built on JavaScript's runtime that uses an event-drivent and non-blocking I/O mechanism [6]
repository	A place where data is being stored and managed
SOLID	Abbreviation for: Single Responsibility, Open-Closed, Liskov substitution, Interface Segregation and Dependency Inversion, are the 5 basic principles for object-oriented programming development proposed by Robert C. Martin [7, section II]
storyboard	A visual presentation for screens inside the application and connections between them [8]
Trait	A design pattern often used in object-oriented programming, whose idea is to provide a set of functions with default implementation so that a class can use to extend its behaviours
unit testing	A software testing method in which a single logical unit is being tested to address potential issues

1 Introduction

According to a monthly report conducted by Statista.com[1], as of June 2016, there have been around 2 million applications in Apple App Store [9]. Compared to June 2015 when there was approximately 1.5 million units, it sees a 33% growth [10]. The market is gradually growing and is expected to double in size by 2020 [11].

The promising raise of iOS applications market results in a tougher battle for market shares. As pointed out in a white paper conducted by Intelliware, there are a number of critical factors that can make up a successful application [12]. Contributing to the likes of business value delivery, branding protection are application technical sides: optimization, scalability and extensibility. Undoubtedly, application growth is always an important element and needs to be arranged carefully during early stage of development. Moreover, Robert C. Martin once claimed that a software project's ability to respond to change plays a vital role in its success [7, p.32]. This implies the importance of constructing the application based on a testable, scalable and maintainable architecture [13].

Over the years, the Model View Controller (MVC) architecture has been a popular choice among iOS developers. As stated by Apple, MVC is a good central design for iOS applications. It offers reusable codes, better defined interfaces and most importantly, a great cooperation with existing Cocoa technologies. However, the strict coupling between MVC's modules often poses a considerable drawback in the reusable aspect. [14] This problem is described as: when more data and business logic arrive, the View Controller module will be overloaded, neither reusable nor testable since it cannot be separated from the View. Thus, MVC is often unabbreviated as Massive View Controller[2]. As a result, a better defined pattern namely Model View View Model (MVVM) is introduced.

MVVM offers an independent business domain called View Model which handles all the data transformation and operations. Since it is a standalone domain and not attached to any other layers, unit testing is easily integrated. Moreover, with the emerge of Pro-

[1]https://www.statista.com

[2]This comes from the fact that View Controller is responsible for many tasks, hence grows expeditiously when the project evolves.

tocol Oriented Programming (POP) in recent years, View Model becomes more flexible and reusable when paired with a Trait-like design pattern [15]. This study aims to demonstrate the use of the Protocol Oriented Model View View Model (POP MVVM) architecture in developing a real-world iOS application with the case study of Tintm[3], a news aggregator application. Moreover, a number of practices that serve as illustrations for software development principles like SOLID and FIRST will also be integrated in this study.

[3]https://www.tintm.com

2 Theoretical Background

2.1 Model View Controller (MVC)

First introduced in Smalltalk[4] in the 1980s, MVC quickly emerges as one of the most influential design pattern for rich graphical user interface application [17]. In iOS MVC design pattern, objects in the application are grouped to 3 different layers: Model, View and Controller. They are separated from the others by some abstract borderlines. Communication between these layers is accomplished through these boundaries [14].

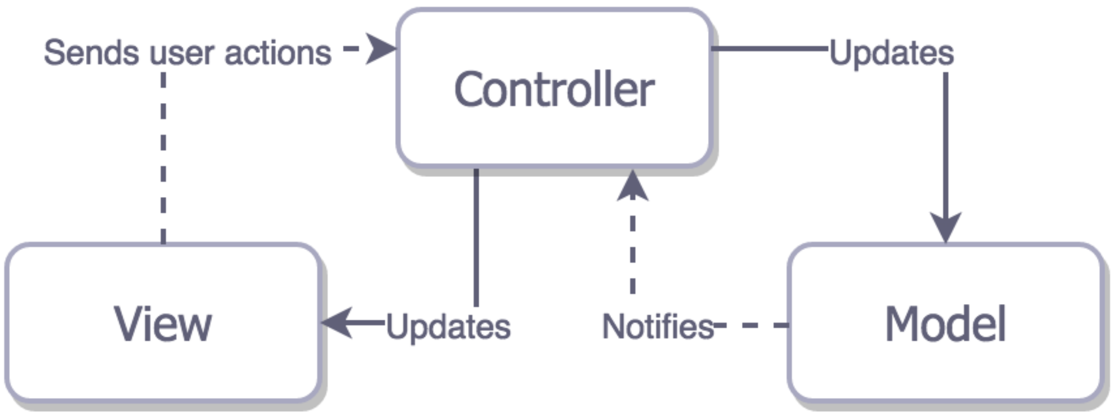


Figure 1: MVC pattern (Reprinted from Orlov (2015) [18])

Data encapsulation, computation and manipulation are done within the Model object. Persistent data and networking data also reside here. Ideally, Model must not have a direct connection with the View layer, since it should not be involved in user interaction operations.

View is the interaction point between users and the application. Its main responsibility is to display data stored in Model and enable editing for that data. Ideally, Model and View will communicate through a Controller as described in figure 1. Nevertheless, as mentioned in chapter 1, they are often coupled.

Controller serves as the middle man between one or more models and one or more views.

[4]One of the very first programming languages that support OOP paradigm, developed around the 1970s for educational purpose [16]

As described in figure 1, there is a ruleset that indicates clear connections between specific domains. As an intermediary, Controller is the conduit through of the pattern in which changes in View or Model are learnt by the other. User interaction made in View is interpreted by Controller which then notifies the changed state in data to the Model layer. Vice versa, any changes occur in Model object is recorded by Controller via Key-Value Observing (KVO) system [19], which in turn updates the data displayed by View. [14]

2.2 Model View View Model (MVVM)

In mainstream Apple's MVC, almost all the business logic is put into the Controller's body. Business logic often consists of: data operations on models, networking tasks, handlers for view's interactions, etc. In some cases, a part of the business logic can be offloaded to Model layer (data calculation for example). On the other hand, the View's only responsibility is to send actions to the Controller. Thus, it makes the overall architecture imbalanced in term of responsibility division.

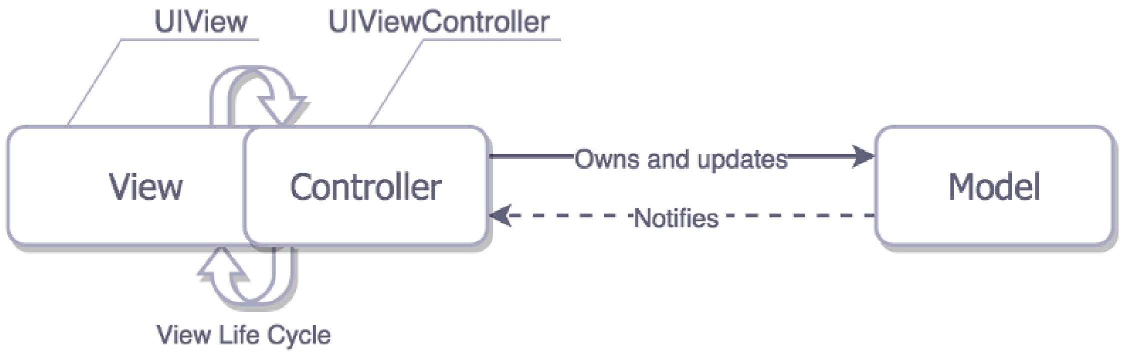


Figure 2: MVC in reality (Copied from Orlov (2015) [18])

In a real-world MVC-based project, View is usually found to be coupled with its Controller as presented in figure 2. This coupling makes both component neither reusable nor testable. Moreover, Controller is easily overloaded and increases steadily in size when the application grows. Addressing the problem of overwhelmed Controller in MVC, Microsoft introduced the MVVM design pattern [20]. Theoretically, MVVM is an upgrade of Apple's MVC, where the main components are kept and the domain logic is delegated to a new layer, namely View Model.

The new states of View and Controller are illustrated in figure 3. They pair up and eventually become a unique domain. Controller does not hold reference to Model anymore, instead they communicate through View Model, the newly introduced layer. Having View Model as a middle man brings many benefits in terms of testing and expanding the application. Firstly, View Model forms a one-way communication with View by binding itself to the corresponding View. As a result, the View layer holds reference to its View Model but the View Model is restricted from knowing about its View. Thus, it can be widely reused and adopted by many different View objects.

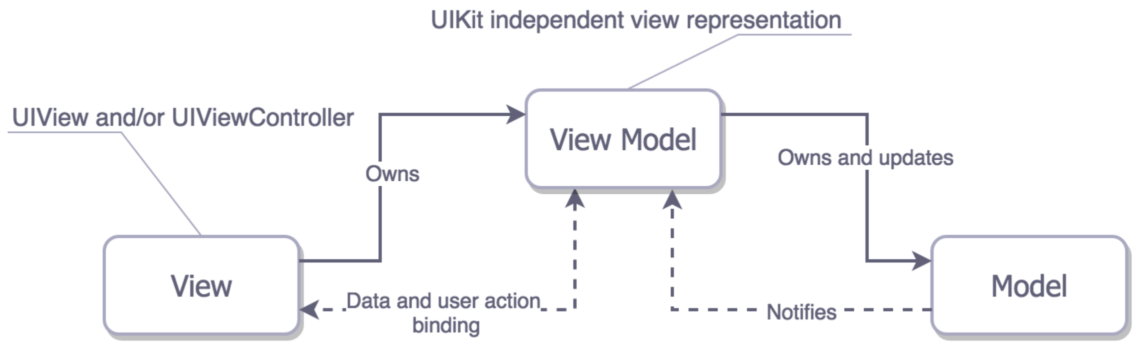

UIKit independent view representation
UIView and/or UIViewController
View Model
Owns
Owns and updates
View
Model
Data and user action binding
Notifies

Figure 3: MVVM pattern (Copied from Orlov (2015) [18])

Secondly, View Model is capable of observing changes in Model and synchronize itself with the events. Since there exists a binding between View and View Model, the former is therefore updated accordingly. This results in a responsive experience for users as well as forming a solid signalization system, where events occur inside each components are decoded as signals and bound to specific handlers. [20]

Finally, unit testing in MVC is often an issue. Most of the business logic code usually resides within the View Controller which makes it the only possible place to perform unit testing. A View Controller's state changes throughout its lifecycle and the same goes for the Model it possesses. When the Model's state is dependent on the UI environment, unit testing becomes trickier since side-effects is bound to happen during the test therefore leaving the test output unverified. Consequently, additional step is required in order to isolate the Model's state so as to prevents UI environment from changing it.

However, in MVVM, View Model is an UI-independent layer. Reference to Controller is eliminated and as a result, Model remains isolated. Performing unit testing upon View Model is thus made easier. Also, little effort is invested in preparation and neglecting side-effects.

2.3 Protocol Oriented Programming (POP)

POP is a programming paradigm which was introduced by Dave Abrahams at the World-wide Developers Conference (WWDC) in 2015 [21]. In contrast to Object Oriented Programming (OOP) whose idea is based on the concept of object[5] [22], POP focuses on the use of protocols. POP in fact is not considered as a preferable paradigm for Swift but more of an enhancement for OOP. Its adoption helps solve many disadvantages OOP has long introduced as well as playing a vital role in improving code base flexibility, scalability and testablity that many OOP-based Swift projects lack.

2.3.1 Protocol

A protocol in Swift is an interface that provides specific methods. Any object conforms to that protocol must implement the given set of functions. According to Apple's documentation: "A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality" [23]. The takeaway point is that a protocol is a blueprint, that is, something that gives instructions but no particular implementation. In other words, a protocol represents actions, indicates how an object can act. As a result, it provides developers with a convenient method of extending an existing object's behaviours, while reducing the overhead of expanding base classes' functionality through the act of inheriting. Moreover, protocols can be widely adopted by classes, structures and enumerations [24, p .41] thus simplifying writing codes in many cases. For example, any class that conforms to an Error protocol could provide its own implementations for error handling.

[5]An unit that encapsulates data and owns the corresponding accessors for that data.

```swift
enum SignupError: Error {
    case invalidPassword
    case invalidUsername
    case emptyField
}

do {
    try signUp(username: "Nguyen", password: "abcd")
} catch SignupError.invalidPassword {
    print("Password must be longer than 8 characters.")
} catch SignupError.invalidUsername {
    print("Username is alreadyTaken.")
} catch SignupError.emptyField {
    print("Some fields cannot be empty.")
}
```

Listing 1: Example of using Error protocol in Swift

Characterizing errors by grouping them into enumerations greatly enhances code readability. Moreover, not only objects can supply their own error handlers as shown in listing 1, but they can adopt the default implementations provided by protocols they conform to. This scenario will be explained in section 2.3.2.

2.3.2 Protocol Extension

In many programming languages, the protocol is restricted from having its own implementation. In contrast, Swift allows a protocol to fulfill its methods by using extension. Not only extension can add functionality to protocols, the same applies to classes, structs and enumerations [25].

By extending a protocol, all of its conformance will be eligible to use the default implementations defined inside the extension. As mentioned in section 2.3, POP favours the use of the protocol over the class. Hence, POP-based projects are often built upon the foundation of protocols. With extensions, protocols can grow and expand without paying extra attention on the compatibility of existing codes. Furthermore, any types that conform to such protocols will therefore take affect as a whole, reducing the need of modifying implementations individually when requirements change.

2.3.3 Trait-like pattern in POP

Trait is a concept that has been adopted by many programming languages with the likes of Scala, Groovy, Hypertext Preprocessor (PHP)[6], etc. Not until the introduction of protocol extension that Trait officially becomes available in Swift. The spirit of Trait lies behind the idea of extending a class functionality without having it inherits from some other classes. This feature proves to be powerful in many situations, especially in the cases of programming languages that do not support multiple inheritance. The comparison between a trait and a protocol is displayed in figure 4.

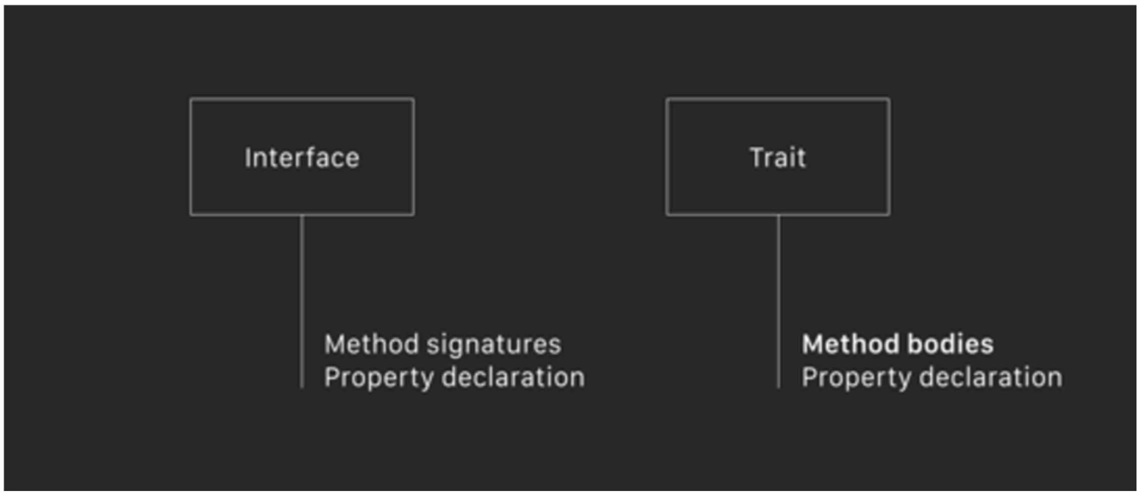


Figure 4: Interface vs Trait (Modified from Matthijs Hollemans (2015) [15])

It is noteworthy that while interface only provides class with method signatures, Trait also offers the method bodies as presented in figure 4. Naturally, using the protocol with extension is the way Trait-like pattern is applied in Swift. As a pattern that supports factoring hierarchies and encourages code reusability [29], Trait is an essential part in an POP based application.

2.3.4 Using Protocol Oriented Programming (POP) in Swift

Given that POP is still a relatively new paradigm, one can ponder over the practical use of POP in Swift development. In fact, there are numerous reasons to support the idea of POP. Firstly, most of the times, it is good practice to adopt the ideas that are implemented

[6]Eventually, Trait is reserved as a keyword in those languages [26–28].

at the heart of a language. As claimed by Dave Abrahams at the WWDC in 2015, Swift is a protocol-orieneted language [21]. An observation on the distribution of classes, structures and enumerations inside Swift's standard library will clarify that claim.

```
Nguyens-MacBook-Pro:swift_stdlib-master nguyenluong$ grep -e
↪  "^protocol" stdlib.swift | wc -l
      73
Nguyens-MacBook-Pro:swift_stdlib-master nguyenluong$ grep -e
↪  "^struct" stdlib.swift | wc -l
      87
Nguyens-MacBook-Pro:swift_stdlib-master nguyenluong$ grep -e "^class"
↪  stdlib.swift | wc -l
       4
```

Listing 2: Using Bash script to display number of protocols, struct and class used in Swift standard library

As observed in listing 2, there are 87 structs, 73 protocols and only 4 classes reside in Swift's standard libraries. This result clearly reaffirms Abrahams' statement on the paradigm Swift is following. Moreover, one big drawback of OOP is the scalability of its inherent system. An OOP's based application is built upon classes and inheritance structure. When a class inherits from another, it has the rights to use the properties and methods defined in the base class and therefore reducing the need of creating its own implementation. However, it is worth noticing that in a majority of programming languages a class can only inherit from a single base class. Thus when the project grows, there appear three possible problematic consequences:

1. Base classes are overloaded with additional functionality if no new class is created to share the responsibilities.
2. New classes are added and the hierarchy tree will grow even deeper hence reduce abstraction level.
3. In many cases, classes may have the behaviours of two or more base classes. Inheritance limits their chance to pick up implementations from other classes.

To illustrate the aforementioned claims, the study will demonstrate a case study taken from an article written by Matthijs Hollemans [15]. This case clarifies the beneficial factors of applying POP concept into a Swift project by analysing the state of a project before and after POP is involved.

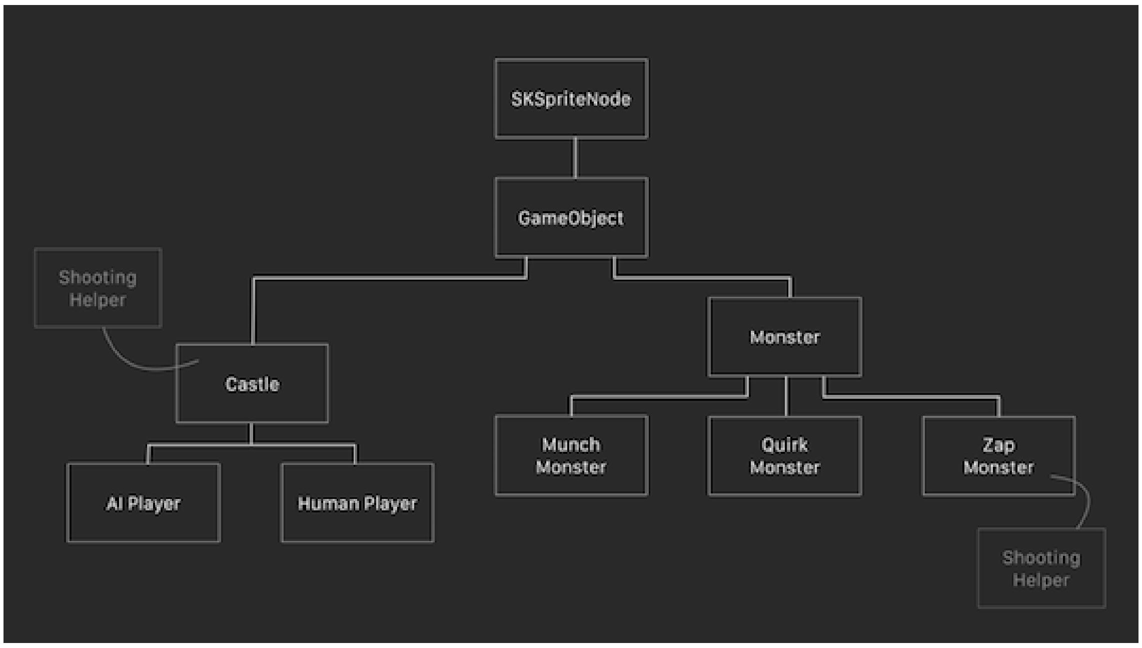


Figure 5: Hierarchy tree of an iOS game (Copied from Matthijs Hollemans (2015) [15])

The hierarchical structure of an iOS game is illustrated in figure 5. It contains a SKSpriteNode class which acts as the root of the hierarchy. Furthermore, there exists multiple classes that inherit from their descendants. Additionally, there are two helper classes responsible for enhancing the class' default behaviour. As observed from figure 5 , the hierarchy tree can grow to a complex stage and become divergent along the growth of the project. Moreover, considering a situation where a newly added class requires behaviours of both Castle and Monster class, it results in difficulty regarding the choice for base class. To overcome such problems, POP servers as a handy tool. By making every class conform to protocols, the deep, nested, complicated hierarchy tree in 5 is flattened into a clean and simple one as described in figure 6.

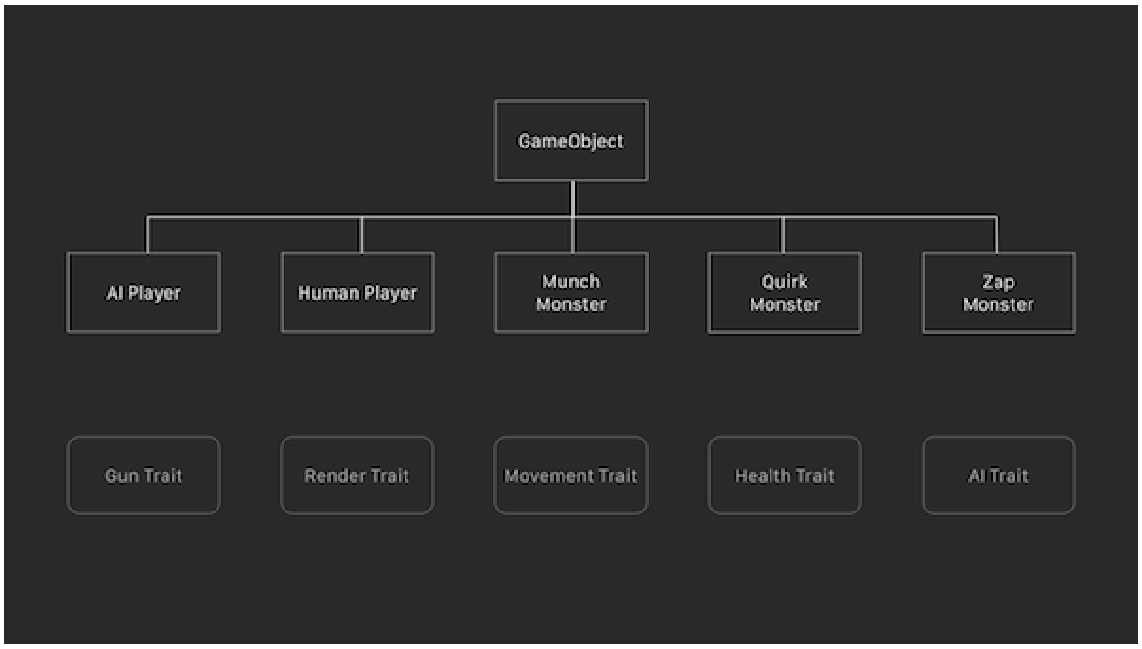


Figure 6: Hierarchy tree of an iOS game after applying POP (Copied from Matthijs Hollemans (2015) [15])

The original idea of POP[7], which is to reduce the need of inheritance as much as possible, is well displayed in this situation. Firstly, relative behaviours are grouped into protocols. Then, the protocols are bound to their own extensions. Finally, default implementations are generated within the extension, thus form multiple Traits. Classes can now choose which types of behaviours they want to own instead of being restricted by the set of actions that their parent classes provide. An explanation for this transformation is described in the following order.

A Munch Monster can move and shoot, a Zap Monster can move and shoot too but a Quirk Monster can only move. Unfortunately, as of figure 5 shows, the Monster class they inherit from only offers the movement ability, which means that Zap Monster and Munch Monster class will have to implement the shooting ability on their own. Another possibility is that the shooting ability can be added directly to the Monster class. However, the addition becomes unnecessary because the Quirk Monster class does not need it.
By creating Traits for different abilities, the Monster's child classes can select the types of Trait they want to conform to. Given that each trait comes with their own implementations, the conformance would require little to no additional specifications for their behaviours. As a paradigm that favors composition over inheritance, POP is a great enhancement

[7]See section 2.3

for the classic OOP, especially in term of neglecting code's redundancy and inheritance complexity in development.

2.4 Protocol Oriented Model View View Model Architecture

One considerable advantage when adopting the MVVM structure is the isolation of business logic. However, in reality, it is difficult to observe this benefit due to the specific binding between Controller and View Model objects. By definition, View Model is a unit that packs all domain logic a Controller needs. Henceforth, a View Model is only reusable in the case where there are 2 or more Controllers share a same package of functionality, which rarely happens in a real-world project. However, this problem can be solved by having View Model conforms to a number of different Traits.

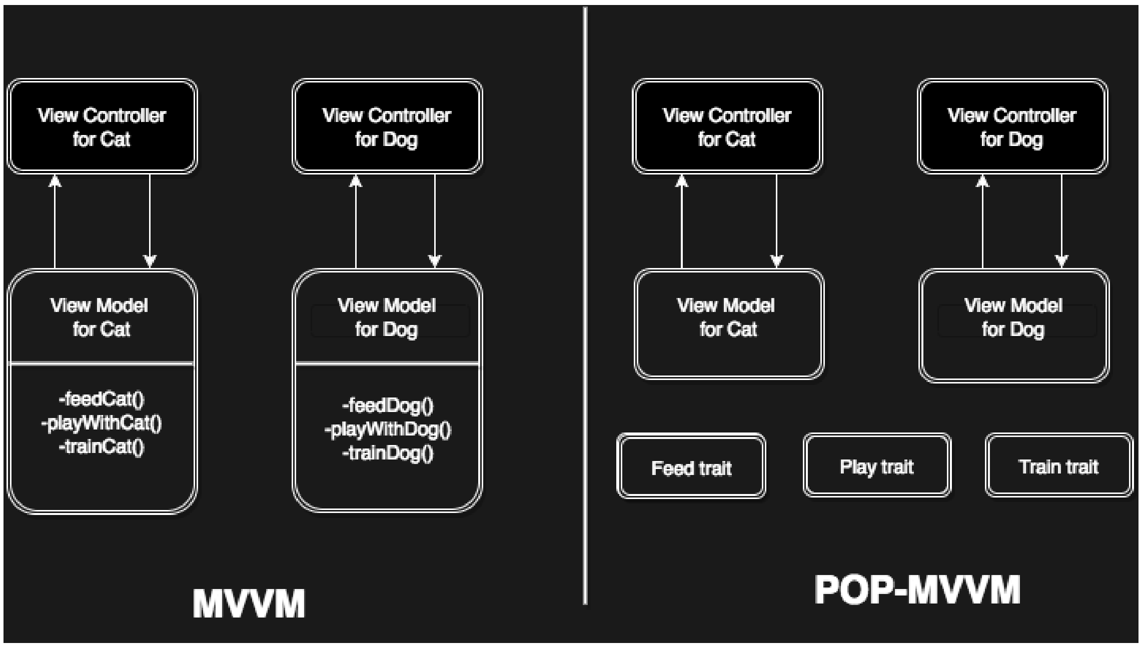


Figure 7: Comparison of MVVM pattern and POP MVVM pattern

Views are unique in terms of responsibilities and the same applies to their View Models. Although it is very unlikely to observe two identical View Models, having two or more of them sharing some functionality is common. If these View Models have completely separated implementations, they may violate the Don't repeat yourself (DRY) rule[8], thus reducing code reusability. The View Models in figure 7 encapsulates interactions one can

[8]The rule states that: "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." [30, p. 46]

have with their pets. However, either it is a cat or a dog, the way they are fed or played with appears to be the same. Consequently, it is not efficient to write a same logic twice.

The right-hand side of figure 7 displays the state of the project after POP is applied. These interactions are split into multiple traits, e.g: Feeding Traits, Playing Traits, etc. Instead of defining their own implementations, View Model can conform to these Traits. This practice greatly reduces the number of duplicated implementations in View Models since each Trait has their own logic defined already. Furthermore, POP considerably enhances the code structure in term of flexibility. For example: number of interactions with the pets may increase in the future. The owner can have them clean the home, feed the kids etc. Basically, without POP, these interactions need to be redefined in each and every View Model. Behaviour Traits provided by POP not only avoids this hassle but also provides foundational implementations that new objects can easily make use of.

3 Application

3.1 Description

The project was carried out on request of SmartApp, a startup company based in Ho Chi Minh City. It was divided into two parts: front-end application and server side implementation. I was in charge of implementing the iOS application while the server-side part was done by another developer. The application flow is displayed in figure 8.

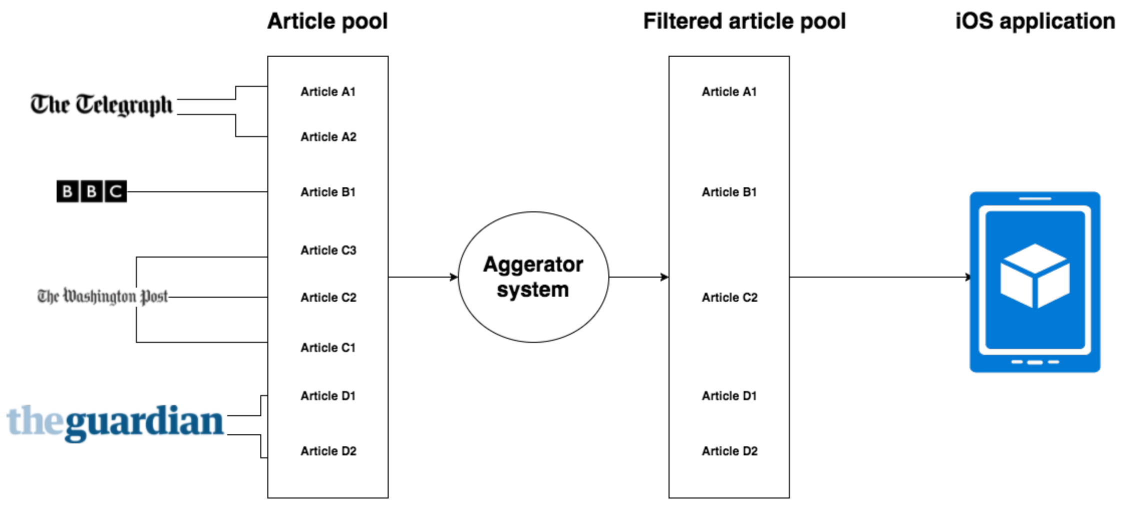


Figure 8: Tintm's application flow

The application aims to present its users the most popular news. For that reason, it is essential to construct a logical and efficient information filtering system. Firstly, articles are gathered from multiple reliable newspaper and sent to a news aggregator as observed from figure 8. Then, inside the system, they are sorted and ranked based on multiple factors, for example: user views, social media interactions, published date, etc. Finally, the system outputs the refined data which will be displayed in an iOS application. User interactions with the application are recorded and sent back to the aggerator system. Based on the records, the system can learn more about its users, study their behaviours and criteria. Consequently, its ranking mechanism improves gradually over time and serves better news which closely suit users' needs.

3.2 Development Process

At the beginning of the development process, the team decided to use Waterfall methodology for this project. As a sequential model, each step in the Waterfall process must be fulfilled before the next step can begin. Furthermore, each step must also be verified to make sure that it complies with the requirements. [31] The process is straightforward and contains no iteration which makes it a viable choice for short time-span project like this (3 months).

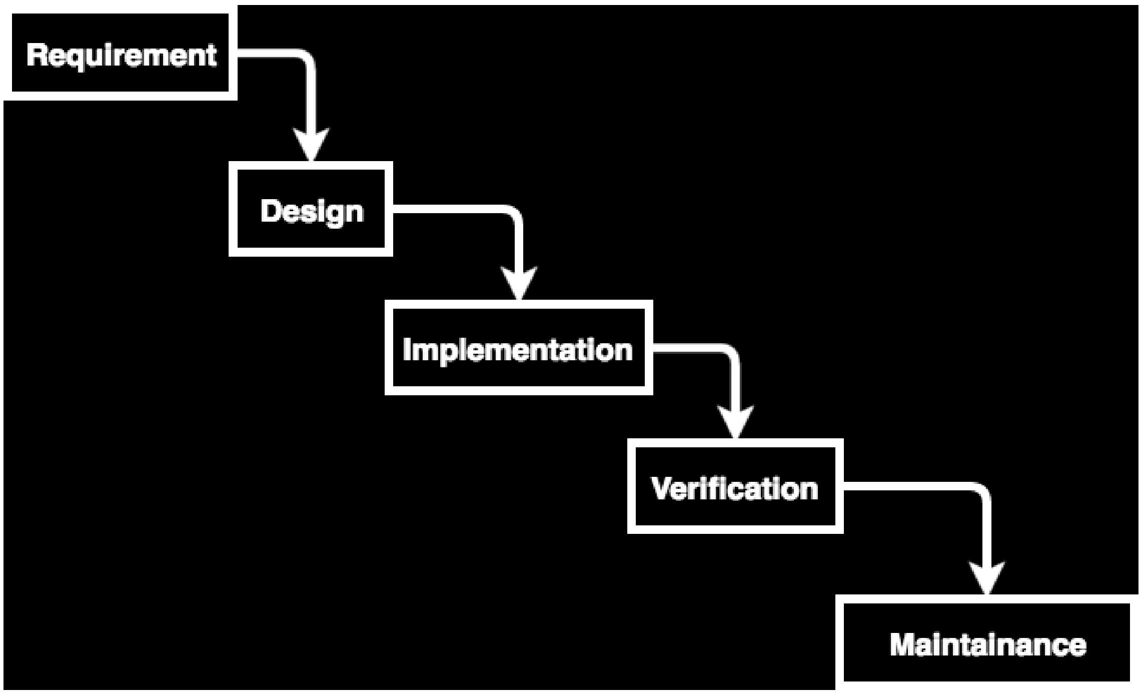


Figure 9: Waterfall process

As seen in figure 9, a typical flow of Waterfall methodology often contains 5 stages. The first step was conducted in the first two weeks of the development process. In Waterfall methodology, it is essential that requirements are clear, concise and well understood by all the parties involved. Accordingly, the planning phase often takes more time than in other methodologies. [31] The design phase was carried out with the involvement of me and a designer and was completed within a week. The implementation phase was planned to be fulfilled in 2.5 months but unfortunately the deadline was not met due to some severe bugs. As a result, the development time had to span to a total of 5 months. The implementation phase took an additional month while verification and maintenance required another month to due. After that, project maintenance was carried out monthly and verified by the team leader.

3.3 SOLID as a Design Guideline

As addressed by Robert Martin, one of the main causes for rotting software is constant requirement changes. These changes are often made quickly by the developers who are not familiar with the design philosophy. Thus over time, the initial design is violated and software quality starts to degrade. [7, chapter 7]

However, changes are unavoidable in software development and should not be the one to blame. An ideal design needs to be non-volatile and remains clean as changes come in. Moreover, it should be well protected from rotting and flexible enough for continuous changes. The practices to achieve such system are encapsulated in SOLID principles. [7, chapter 7] With the hope of reducing code complexity, keeping a cohesive design and maintaining a robust code base, the application adopts SOLID principles as its system design guideline. It involves classes and functions design , frequent integration of unit tests, etc.

3.4 Technologies

The client application was written in Swift. Although Objective-C is also a reasonable choice for iOS development, Swift was the more favorable for this project. Reasoning for this choice is carried out in section 6.2.

The backend application was built by Node.js. Renowned for the ability of building fast and scalable server application, Node.js is a reasonable choice for server-side building. Taking advantages of its non-blocking mechanism (also known as event-driven architecture)[9], Node.js effectively boosts its performance as well as optimizes its throughput. [33] Also thanks to the non-blocking I/O model, Node.js can concurrently handle multiple connections, make it an excellent candidate for data-intensive and real-time applications [34]. A simple benchmark between Node.js based server and PHP based server is taken to jus-

[9]A mechanism that allows codes to run without blocking execution of others. Detailed explanation is located at Node.js official documentation [32].

tify the aforementioned claims. Its result is illustrated in figure 10.

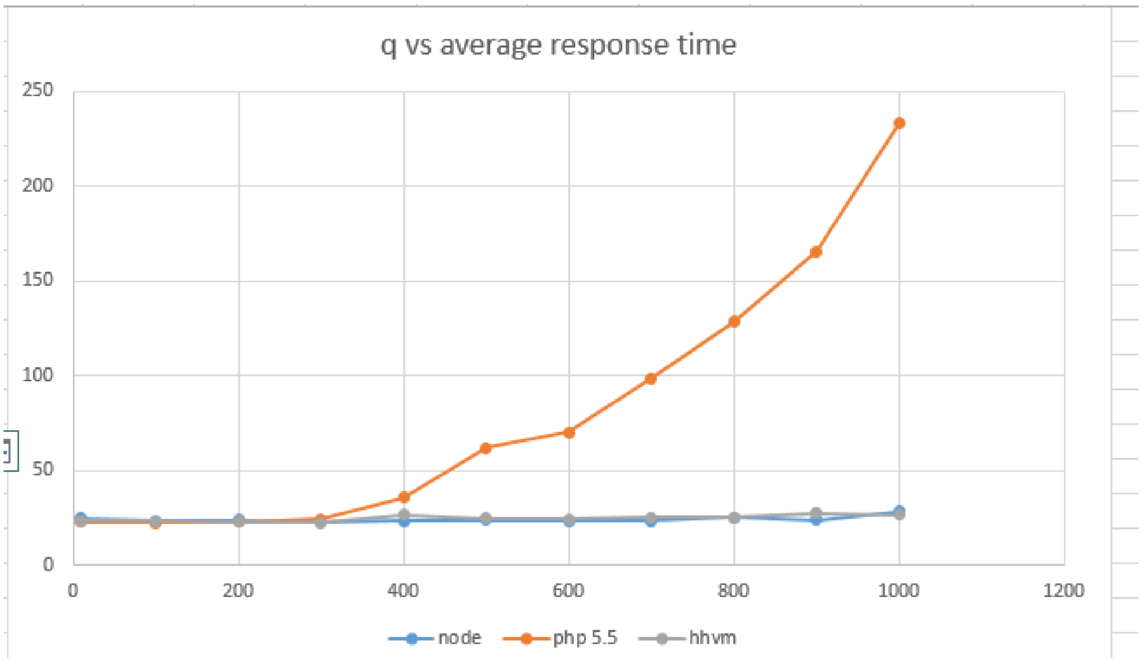


Figure 10: Node.js vs PHP Hypertext Transfer Protocol (HTTP) and Central Processing Unit (CPU) benchmark (Copied from Roberto Sanchez (2016) [35])

The test aims to compare HTTP and CPU capabilities of both frameworks by making them execute bubble sort algorithm for a definite amount of input. The outcome from figure 10 shows that Node.js' performance is clearly ahead of PHP when the amount of input increases: nearly ten times faster when the input consists of 1000 elements. Since the focus of this study is on the iOS development, Node.js' productivity and performance will not be further discussed.

4 Implementation

4.1 Project Structure

4.1.1 MVVM block

The project structure is built upon the foundation of multiple Model-View-View Model combinations. For better convention, these combinations will be denoted as ”blocks” from now on. Each block strictly follows MVVM design principles [20]. In addition, they also adapt to POP paradigm, thus form a more advanced architecture namely Protocol Oriented Model View View Model (POP MVVM).

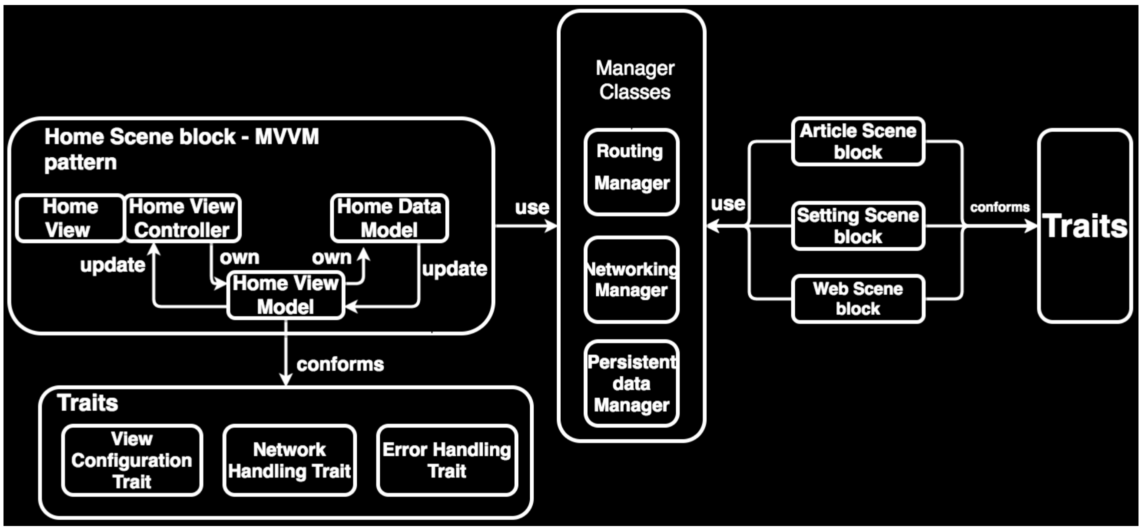


Figure 11: Project structure diagram

The components inside each block establish their connections through a binding mechanism as mentioned in section 2.2. As seen in figure 11, a Model is bound to a View Model. Likewise, View Model is bound to a View Controller. These relationships are called one-way bindings [36], where the inner object does not know anything about its owner. On the other hand, changes made within an inner object will be recognized by its owner through KVO system. Since all units adopt the same architecture, only the construction for Home Scene block will be discussed in detail.

4.1.2 The inclusion of Traits

As described in section 2.2, the workload for View Model appears to be heavy since it is an isolated logic domain. View Model is responsible for processing networking data, exposing required information to View, catching and handling errors, etc. However, the appearance of Traits in this architecture helps offload a significant amount of work for View Model. Traits contain a set of features as well as basic implementations for them. By conforming to these Traits, View Model does not need to setup everything from scratch since they can use the existing solutions or modify them to suit their own needs.

There are a number of different Traits being used in this project. For better convention, they are named after their unique responsibility. For example, a Trait responsible for Label View configuration is called Label Presentation Trait. In the same manner, Trait used for Image View setup is named Image View Presentable. For the sake of this study, only two aforementioned Traits will be discussed in detail in section 4.2.5.

4.1.3 Manager Classes

For better abstraction, this project uses a number of Manager classes. Moreover, since related implementation details are wrapped into respective class and isolated from outside objects, it is easier to investigate bugs or potential flaws in the future. Nevertheless, there exists debates among the use of Manager class in software development. Thus, overusing is not encouraged[10].

In MVC based Swift projects, navigation between scenes is often done within View Controller component [14]. However, it is not the case in POP MVVM. As shown in figure 11, blocks do not communicate directly with the others, instead the communication is accomplished in a Routing Manager. This class takes care of creating necessary View and View Model objects, handling navigation transition animation and performing the navigation. In other words, Routing Manager's main responsibility is to glue all the blocks together. Such setup is inspired by another renowned iOS architecture namely View Interactor Presenter

[10]More information regarding the debate can be found at: https://blog.codinghorror.com/i-shall-call-it-somethingmanager/

Entity Router (VIPER)[11].

Networking Manager plays a vital role in almost every iOS application since it is the bridge between client and server applications. All operations relate to data from remote system should reside here, some of which include: making requests for data, constructing required parameters for requests, processing responses from server, etc. Processed data will be further manipulated inside responsible View Model via a callback method.

Finally, as one goal of this project is to offer users offline reading ability, data persistence needs to be taken into consideration. After fetching articles from server, the application will attempt to store them into device's database. Persistence Data Manager utilizes all device's storage related operations. It can request access to the device's database and perform read, write, update or delete function there. The application uses Realm for data persistence due to its simple and modern features set[12].

4.2 Implementation in detail

As affirmed in section 4.1.1, this study restricts detailed explanations for Home View, Home Controller and Home View Model. The study focuses on the manipulation of MVVM architecture and POP paradigm, therefore, Manager classes implementations will not be described.

4.2.1 Home Scene Description

Home Scene serves as the entry point of the application. It greets users with multiple lists of articles, sorted into respective categories. For better illustration, a blueprint of this scene is displayed in figure 12.

[11]This architecture is complex and only applicable in large scale project. More information concerning its implementation can be found at https://www.objc.io/issues/13-architecture/viper/

[12]https://github.com/realm/realm-cocoa

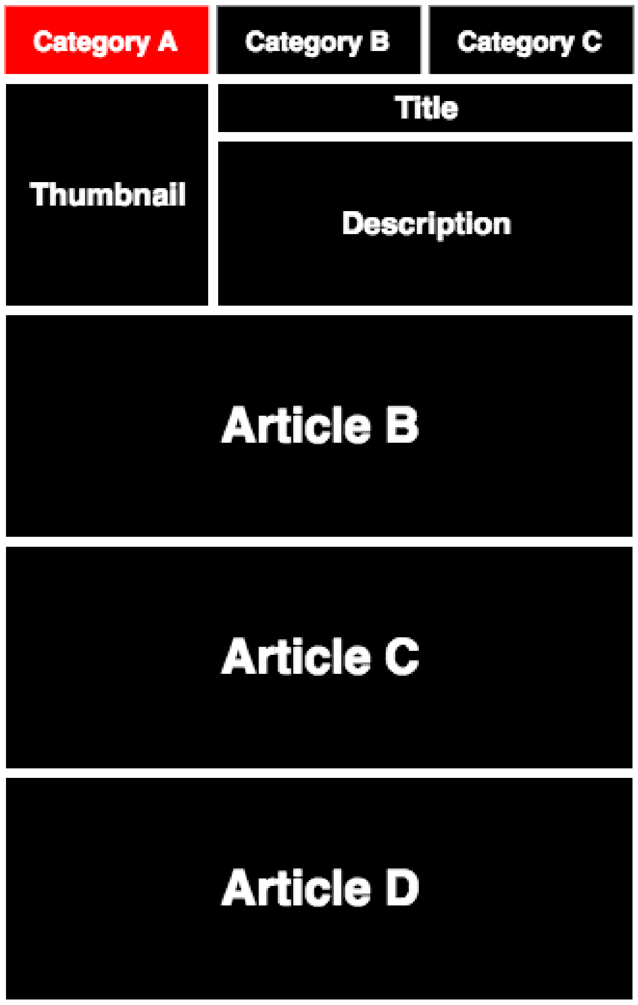


Figure 12: Home scene blueprint

As observed from figure 12, Category A is selected since its background color differs from the others. Therefore, the articles below will belong to that category. At top of this scene lies a navigation bar which represent the list of categories. Users can navigate to different sections by sliding leftward or rightward. Accordingly, the list of articles will also change. Each article contains a thumbnail, a title and a brief description. The design for this scene is kept as minimal as possible so as to improve the overall performance and avoid the user's distraction.

4.2.2 Home Model

The Home Model object wraps up necessary data for an article displayed in Home Scene. A diagram for this class is illustrated in figure 13.

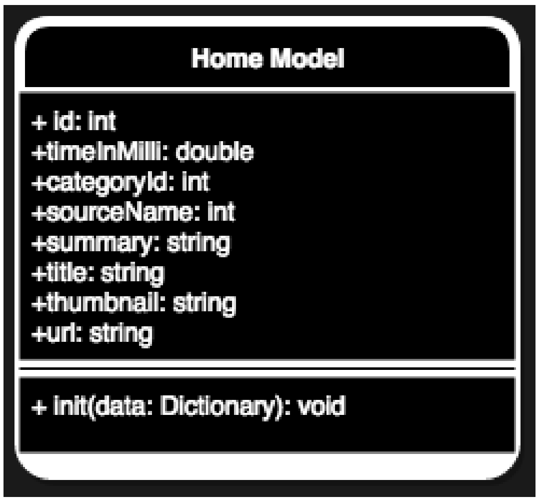


Figure 13: Home model diagram

The parameters in figure 13 are self-explanatory already. Some notable variables are: `id`, `categoryId`, `title`, `thumbnail` and `summary`. First of, `id` variable denotes a unique identification for the article. It will be used to fetch news' content later. `CategoryId` specifies the identification of the category which this article belongs to, for example: Sport category has the id of 1, Politic's id is 2 etc. `title` and `summary` contain the article's headline and brief summary respectively. Finally, `thumbnail` is a URL string which the application uses to download the thumbnail image for that article. Apart from the above variables, there is an initialization method in this class namely `init` whose configuration is explained in listing 3.

```
mutating func mapping(map: Map) {
        id <- map["Id"]
        categoryId <- map["CategoryId"]
        categoryName <- map["CategoryName"]
        content <- map["Content"]
}
```

Listing 3: Initialization method of Home Model

This method processes raw data fetched from server, which is often a JSON string. It does a simple operation: mapping values stored in a JSON string into the object as shown in listing 3. The value of key `CategoryId` is mapped into `categoryId` variable, `Id` is mapped into `id` variable and so on.

4.2.3 Home Controller

Home Controller is responsible for the user interface control. It captures user's interactions and vice versa, reflects changes in Home Model to user. Unlike in MVC where Controller has to provide actions for these updates itself, the handlers are put in View Model. Therefore, Home Controller only has to signal Home View Model about user interactions and requests updated data from it as well.

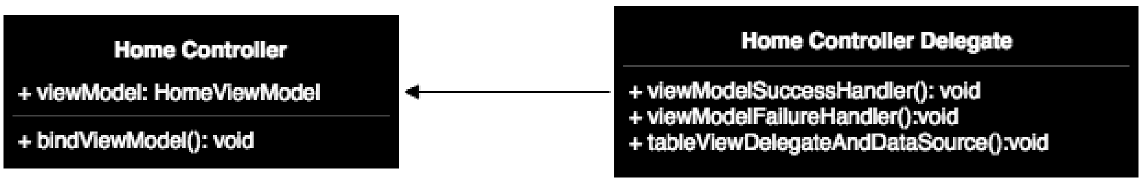

Home Controller
+ viewModel: HomeViewModel
+ bindViewModel(): void
Home Controller Delegate
+ viewModelSuccessHandler(): void
+ viewModelFailureHandler():void
+ tableViewDelegateAndDataSource():void

Figure 14: Home Controller diagram

Home Controller is minimal and its only responsibility is to hold a View Model object as seen in figure 14. Everything relates to UI configuration and View Model conformance is accomplished inside the storyboard or delegated to Home Controller Delegate class. There are 2 Home Controller Delegate components which live within the app but for demonstration purpose, they aggregate to form a single class in figure 14. The intent of splitting Home Controller functionality to multiple Delegate class is to satisfy the Single Responsibility Principle[13]. According to the principle, each class or extension should only be responsible for one functionality, whether it is setting up UI attributes or handling data from View Model.

4.2.4 Home View Model

Being the data repository, Home View Model is in charge of refining data from multiple sources (server, device's database, etc.) as well as performing data binding between Home Model and Home Controller. Additionally, it also holds the necessary configuration for UI components inside Home Controller.

[13]The first principle in SOLID which states that: "A class or module should have one, and only one, reason to change" [7]

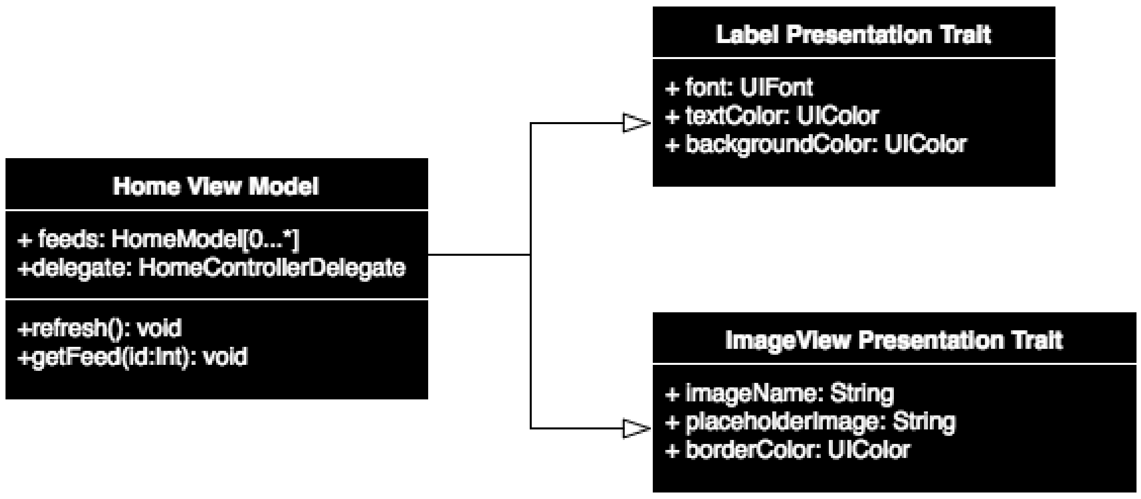


Figure 15: Home View Model diagram

Apart from holding an array of Home Model objects, Home View Model also owns a delegate property as shown in figure 15. This object contains a set of callback functions and acts as a communication protocol between View Model and the Controller it binds to. As illustrated in listing 4, Home View Model wraps up network operations inside the 2 functions namely `refresh` and `getFeed`. The results of these network calls will be handled inside Home Controller as presented in listing 5.

```
class HomeViewModel:LabelPresentableTrait, ImagePresentableTrait{
    //This method refreshes the list of feeds that lies under category
↪   owns the Id
    func refreshFeedsWithCategoryId(id: Int){
        NetworkManager.sharedInstance.getFeedsWithCategoryId(id){
↪   (response) in
                switch response.status{
                case .Success:
                    if let value = response.res{
        //Manipulate the gotten data here before sending them to Home
↪   Controller
                        self.delegate?.successGetFeeds(values)
                    }
                case .Failure:
                    self.delegate?.failureGetFeeds(response.error!)
                }
        }
    }
}
```

Listing 4: View Model functions

Network tasks are often asynchronous and can complete at any time. For that reason, Controller may not be aware of its data's state, so as to accordingly update the UI components. The callback function is the middle man who forms the contract between the Controller and the data it needs.

```swift
class HomeViewController: UIViewController{
    override func viewDidLoad() {
        //Tell View Model that its delegate functions will be handled
↪   here
        self.viewModel.delegate = self

        //Ask View Model to refresh a list of feeds
        self.viewModel.refreshFeedsWithCategoryId(1)
    }

    //Handle the response from the refresh request by providing
↪   implementations for View Model's delegate functions
    func successGetFeeds(values: [HomeModel]){
        //The operation succeeded, now do something with the data
        //For example: reload the table view where the data is being
↪   displayed
        self.tableView.reloadData()
    }

    func failureGetFeeds(error: NSError){
        //The operation failed with an error
        //Print error to console
        print(error.description)
    }
}
```

Listing 5: View Controller functions

As observed from listing 5, Home View Controller contains 2 separated callbacks namely `successGetFeeds` and `failureGetFeeds`. A callback function often carries a status code which identifies the state of a operation, whether it is a success or a failure. A successful operation delivers a set of refined data while a failed one contains an error object. Moreover, callback functions are bound to their respective tasks and require implementations from the appropriate Controller.

The same set of data can be treated differently depending on where they are used. Data inside Networking Manager is considered ”source data” and should remain as-is. Therefore, multiple copies of them are passed to View Models so that they can be freely modified without affecting the real data. As a result, only responses gathered from Network Manager are being manipulated inside View Model, leaving the source data untouched.

4.2.5 Presentation Traits

According to Apple’s design principles, an application should offer consistency in its UI design [37]. Consistency can be achieved in many ways, from providing a default font type for all labels to giving all icons a default tint color. As a result, settings for UI components inside the app may not differ a lot from each other. On the downside, one small change in design requirement would affect the whole system due to the same reason. Furthermore, duplicated settings are unavoidable and inconvenient if there are too many components sharing a same UI setup.

Knowing that each UI component has a set of specific attributes, one can create a Presentation Trait that wrap ups all its visual appearance properties, then equip them with initial values. Arrangement for a Label Presentation Trait setup is illustrated in listing 6.

```swift
protocol LabelPresentationTrait {

    //Color
    var backgroundColor : UIColor { get }
    var textColor : UIColor { get }

    //Font
    var textFont : UIFont { get }

    //Custom
    var numberOfLines : Int { get }
    var lineBreakMode : NSLineBreakMode { get }
}

//MARK: - Default values
extension LabelPresentationTrait {

    var backgroundColor : UIColor {
        return UIColor.whiteColor()
    }

    var textColor: UIColor {
        return UIColor.blackColor()
    }

    var textFont : UIFont {
        return UIFont(name: "Roboto-Bold", size: 15)!
    }

    var numberOfLines: Int {
        return 0
    }

    var lineBreakMode : NSLineBreakMode {
        return NSLineBreakMode.ByTruncatingTail
    }
}
```

Listing 6: Label Presentation Trait source code

A typical Label View often defines attributes for the text it displays. As seen in listing 6, all necessary attributes for a simple Label View are wrapped up in a protocol. Furthermore, their initial values are provided by an extension which together form a Trait with its protocol. The Trait is then picked up by Home View Model and later used for UI configuration for cells displayed in Controller. Configuring duty inside a cell is illustrated in listing 7.

```swift
//Inside HomeCell class
class HomeCell : UITableViewCell {
    typealias Presenter = protocol<LabelPresentableTrait,
↪  ImagePresentableTrait>

    //UI configuration
    func configureWithPresenter(presenter: Presenter){
        textLabel?.font = presenter.textFont
        textLabel?.textColor = presenter.textColor
        textLabel.backgroundColor = presenter.backgroundColor
        imageView.layer.borderWidth = presenter.borderWidth
        imageView.layer.borderColor = presenter.borderColor
    }
}

//Inside Extension of HomeViewController
extension HomeViewController: UITableViewDataSource, ButtonDelegate{
    //Method used to setup cell

    func tableView(tableView: UITableView, cellForRowAtIndexPath
↪  indexPath: NSIndexPath) -> UITableViewCell {
        let cell =
↪  (tableView.dequeueReusableCellWithIdentifier(HOME_CELL_ID,
↪  forIndexPath: indexPath) as? HomeCell)!
        //Home View Model conforms to both Traits, hence it can be
↪  treated as a Presenter
        cell.configureWithPresenter(viewModel)
    }
}
```

Listing 7: Configuration for a cell display

The code in listing 7 suggests the possibility of forming a new type from multiple Traits through the use of `typealias`. Presenter, the newly introduced type, is a compound type composed from Label Presentable Trait and Image Presentable Trait. Any class conforms to both aforementioned Traits can be treated as a Presenter. Additionally, the Presenter type is extensible, meaning that a new Trait can be added or an existing Trait can be removed. In this situation, Home View Model is capable of configuring the Home Cell since it met the conformance requirement and can be treated as a Presenter object.

Furthermore, as also seen from listing 7, implementation details of Home Cell object are not exposed since they are kept within the class' scope. It prevents outside objects from modifying the original class but encourages them to extend its functionality by providing new Traits. This practice implies the definition of the Open/Closed Principle which indicates that a class should be resilient to change, but flexible enough to adapt requirement changes [7, p. 179]. Henceforth, the class is treated as a single module and easily exported for future needs, therefore greatly improves code reusability.

However, one can argue that the configuration task displayed in listing 7 could well be accomplished by making subclasses for both the Label View and the Image View. In compared to the inheritance way, current approach seems to be overcomplicated. On the other hand, it has not fully imposed the essence of POP: composition over inheritance [38, p. 17]. Thus, a further improvement is brought up later in section 6.3.

4.2.6 Unit Testing

Unit testing is essential in application development. However, it is often overlooked by developers and consequently results in error prone, unstable code bases. On the other hand, improper architecture would also lead to difficulty in test arrangement. For example, UI components and business logic coupling makes it difficult to create an isolated test environment. Unit testing in MVVM is made easier due to the isolated domain logic as explained in section 2.2. A simple test case is presented in listing 8.

```swift
func testHomeViewModelFilterArrayBySource(){
    //Models array which will be passed to View Model
    let models =[Model(id:1, source:"VNN"),Model(id: 2 source:"VNN"), Model(id: 3,source:"ABB")]

    //Desired result
    let resultModels =[Model(id:1, source:"VNN"), Model(id:2, source:"VNN")]

    //Initialize view model with the models array
    let viewModel = HomeViewModel(models: models)

    //Test if the model array is valid (no duplicated models, all models have unique ids)
    XCTAssertTrue(viewModel.isModelsValid())

    //Test if the result of the function matches with the result array
    XCTAssertEqual(viewModel.filteredBySources(source: "VNN"), resultModels)
}
```

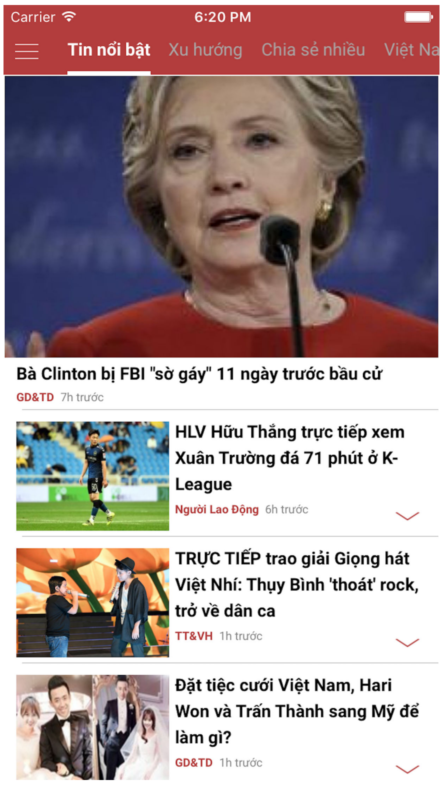
Listing 8: Home View Model's unit tests

The unit test described in listing 8 covers a simple test case for some functions live within Home View Model. As their name suggest, the functions are respectively responsible for validating the models and filtering a list of news, picking out only articles belong to a definite sources. The test prepares a list of mock articles and passes it inside the Home View Model. A sample output is also provided for later evaluation.

In order to succeed, the View Model must sequentially pass two test cases. Firstly, data validation must be done correctly, meaning that input which contains 2 or more identical objects will cause the test to fail. Secondly, the filtered output must meet test's requirement, in other words, output that mismatches the result provided by the test will cause failure. The test is performed in a manner that prevents later assertions from executing in case of failure.

5 Result

The outcome of this study is Tintm, an iOS application already available for sale on Apple App Store[14]. Information regarding the application's services can be found under description section on the same site. The source code is partly open due to contract boundaries, hence not fully available for review.



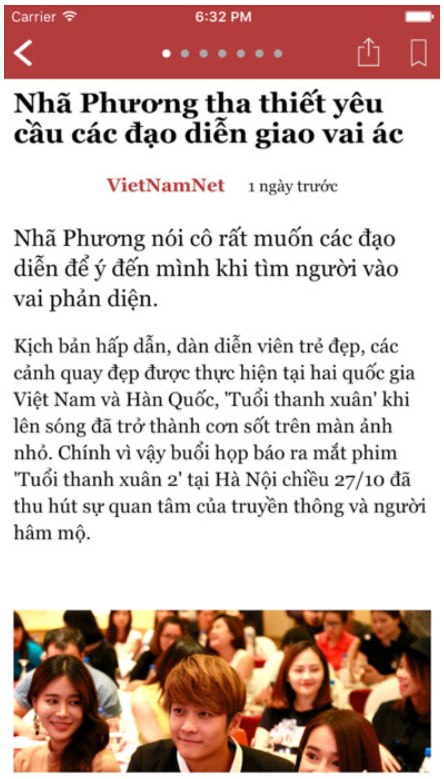
Bà Clinton bị FBI "sờ gáy" 11 ngày trước bầu cử
GD&TD 7h trước

HLV Hữu Thắng trực tiếp xem Xuân Trường đá 71 phút ở K-League
Người Lao Động 6h trước

TRỰC TIẾP trao giải Giọng hát Việt Nhí: Thụy Bình 'thoát' rock, trở về dân ca
TT&VH 1h trước

Đặt tiệc cưới Việt Nam, Hari Won và Trấn Thành sang Mỹ để làm gì?
GD&TD 1h trước

Figure 16: Home view



Nhã Phương tha thiết yêu cầu các đạo diễn giao vai ác

VietNamNet 1 ngày trước

Nhã Phương nói cô rất muốn các đạo diễn để ý đến mình khi tìm người vào vai phản diện.

Kịch bản hấp dẫn, dàn diễn viên trẻ đẹp, các cảnh quay đẹp được thực hiện tại hai quốc gia Việt Nam và Hàn Quốc, 'Tuổi thanh xuân' khi lên sóng đã trở thành cơn sốt trên màn ảnh nhỏ. Chính vì vậy buổi họp báo ra mắt phim 'Tuổi thanh xuân 2' tại Hà Nội chiều 27/10 đã thu hút sự quan tâm của truyền thông và người hâm mộ.

Figure 17: Article view

The application flow is simple, thus creating a user friendly environment. Users are greeted with the home view as seen in figure 16. From there, they can select their favourite news' category, or immediately choose to read an article. Selecting an article from home view will navigate users to its detail page. The detail page, as presented in figure 17, serves the news content and indicates its original source so as to avoid copyright problem. Furthermore, users can navigate to related articles by swiping leftward or rightward. Not only that, sharing and bookmarking are also available. These functionalities can be accessed via the two buttons in the top right-hand corner.

[14]https://itunes.apple.com/us/app/tintm/id1147075214?ls=1&mt=8

The application is in its early stage and thus still open for new features and UI/UX adjustment. For the same reason, feedback is always welcome, as well as bug reporting or recommendations for changes. On the other hand, code base maintenance is done monthly and more improvements will come within the near future.

6 Discussion

6.1 Project outcome

During the development progress, I have been struggling to find a well-rounded resource concerning detailed implementation of the POP MVVM architecture. The subject is really new and has not gained enough attention from fellow iOS developers. Nevertheless, there is a speech worth reading about POP MVVM carried out by Natasha Murashev as well as an in-depth discussion about practical MVVM written by Ash Furrow [39; 40]. They together deliver a starting point for further study of POP MVVM. Also, they imply the importance of crafting a testable application since it is the matter that is often overlooked by the majority of developers.

From a technical perspective, the project met its initial goal, which was to build a POP MVVM based application. The architecture seems to be promising since the system remains clean and cohesive after multiple iterations of changes. Different maintainers have got their hands on the project and the overall feedback is positive[15]. However, since the project is in its early stage, beneficial sides of the architecture are difficult to be recognized.

The use of POP for this project was limited within UI-related duties. However, there are numerous occasions where it can be applied, especially in the case of networking tasks. MVC and MVVM share one weakness: neither of them defines where the logic for network code should go [40]. POP copes well with this problem and is a potential enhancement for existing MVVM-based apps [41].

On the other hand, there are definite drawbacks. Firstly, the application is using a number of third-party libraries. It is the sign that the application is strongly entangled with the libraries' Application Programming Interfaces (APIs). Thus, the former has to catch up with the latter's evolutions. This process may pose multiple problems including: backward-compatibility, security flaws and potential bugs. Hence, special consideration for third-

[15]Quang Vo: "The intention for each layer is clear and straightforward. It is also easy to apply unit test and UI test."

party libraries usage is advisable. [42]

Secondly, as mentioned before, POP MVVM is rather an immature architecture. As of the time of this writing, it has not built enough fame so as to replace MVC as the principle model for iOS development and mobile development as a whole. The use of POP MVVM in this project is purely out of my interest in learning, thus not recommendable as an architecture choice for real-world applications. Especially in the case of inexperienced developers, POP MVVM should be considered an advanced architecture since it requires plenty of time to get familiar with Swift, programming paradigms and effective use of protocols and extensions.

Finally and most importantly, the application is built in Swift 2.0. In recent years, Apple has put great focus on the evolution of Swift. Consequently, the community has witnessed the arrival of Swift 3.0 in June 2016[16] and at the time of this writing, Swift 3.1[17]. Thus, migration is a special concern since the project is no longer compatible with recent iOS versions.

From a non-technical perspective, the application is fully functional and critical bugs are yet to be found. However, the growth rate does not seem to be encouraging (40 downloads during the first 2 months). This opens up questions concerning the current services, UI/UX design and even a marketing campaign.

6.2 Swift over Objective-C

Nowadays, when it comes to iOS development, developers often have to make a choice between the two programming languages namely Objective-C and Swift. Before WWDC 2014, when Swift was still in a beta version, Objective-C would make an obvious choice. However, the situation has changed rapidly since the introduction of Swift at the WWDC 2014[18]. Acclaimed by Apple as a fast, powerful and safe language [24] , Swift has step by step replaced Objective-C as the most popular programming language for iOS development. A comparison upon searched term's trend between Objective-C and Swift is demonstrated in figure 18 in order to make clear of the aforementioned claim.

[16]https://swift.org/blog/swift-3-0-released/
[17]https://swift.org/blog/swift-3-1-released/
[18]https://developer.apple.com/videos/play/wwdc2014/402/

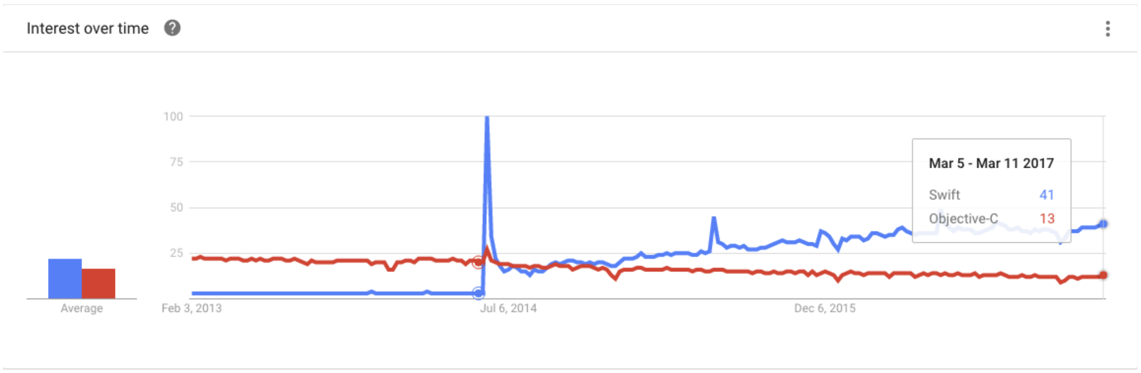


Figure 18: Google trend for searching terms related to Objective C and Swift from Feb 2013 to March 2017

The rapid growth of Swift after 2014 can be observed from figure 18. Its interest index hit the peak on July 2014 when Apple first introduced Swift at WWDC and has gradually risen since then. As of the time of writing, Swift possesses an interest rating of 41, three times better than its predecessor. Nevertheless, Swift's powerful and modern set of features is what truly makes it the language choice for this project. Not only Swift eases the developer's life with its interactive environment and errors control system, it also shortens the development span by providing compact syntax and naming convention which leads to better code's readability and reusability. [43]

Lastly, with the recent addition of Protocol Extension[19], Swift officially becomes a POP supporting language. Since the main point of this study was to demonstrate the practical use of POP and MVVM architecture in a real-world project, Swift was an obvious choice over Objective-C.

6.3 Presentation Traits - A Better Approach

Protocols are blueprints, well-defined contracts and, foremost, abstract instances. They can be easily injected into or ejected from concrete instances. Thus, they should be thought of as pluggable components. In listing 6, this characteristic is not demonstrated clearly. Label Trait is a big block and could be still be broken down into several pieces. By further dividing Label Trait into multiple smaller protocols, the code's abstraction level improves greatly. Not only the newly formed protocols can group up to form the original

[19]Only added since the release of Swift 2.0. Release not is available at: https://developer.apple.com/swift/blog/?id=29

Label Trait, they can also be combined with other protocols to form new Traits if needed. This technique is displayed in listings 9, 10 and 11 in appendix 1.

Regarding the code’s readability, POP is also a great enhancing tool. As mentioned in section 2.3, the protocol stands for behaviour. On the other hand, the protocol is verbose, hence its intention is clear and meaningful. By investigating an object and the list of protocols it conforms to, the developer could grab its capability quickly. Perhaps more interestingly, 2.3 is capable of transforming UI traditional configuration into a CSS-like styling. The source codes listed in appendix 1 also demonstrate this progress.

7 Conclusion

Proposals for scalable, maintainable and testable architectures have always sparked broad interest among developers. Concerning this point, Model View Presenter (MVP) and VIPER are also interesting topics that have been discussed in a number of articles[20]. From my point of view, a flawless architecture does not exist since they all pose strengths and weaknesses. Additionally, the choice for the architecture should be made unbiased and dependent on specific situations. For instance, although mainstream MVC is described as an outdated architecture within the scope of this study, it can still scale up well and be testable if configured correctly[21].

Although the study proposes the practical use of POP MVVM, it does not encourage its readers to hastily apply the model in a real-world project since it is a rather immature and complicated architecture. In my opinion, applying a complex architecture into a real-world project does not necessarily prove one's skill in software development. Rather than that, it is the ability to carry out correct decisions on the use of architecture in different situations. Using a poorly scalable architecture in a large project often results in an unmaintainable code base. Meanwhile, applying a complex model into a simple application is an overkill and not worth the effort spent. All in all, it is not the architecture that prevents its users from testing, scaling or maintaining. The outcomes all lie on the developers' choice.

On the other hand, this study encourages further reading on the following software principles: SOLID principles and FIRST principles. This project could only partly follow the guidelines provided by them, mostly due to my inexperience. However, it proves to be worth the effort since the outcome is a clean, easy to maintain and well documented code base. There are currently a number of articles concerning the practical uses of these principles[22,23].

Last but not least, state of the art is challenging, but not impossible. It cannot be achieved

[20]https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52
[21]An example for modern MVC base app is available at: https://www.raywenderlich.com/132662/mvc-in-ios-a-modern-approach
[22]https://github.com/ochococo/OOD-Principles-In-Swift
[23]http://agileinaflash.blogspot.fi/2009/02/first.html

in a day or two, or by a single person. Thus, this study strongly proposes further exploration into the developing and customization of POP MVVM, especially in the Swift language. In the end, it is all up to one's curiosity and creativity that define the characteristics of his/her own architecture.

"True art is characterized by an irresistible urge in the creative artist."
— Albert Einstein

References

1 Martin RC. Clean code. A handbook of Agile Software Craftmanship. Prentice Hall; 2009.

2 H James de St Germain. Interfaces;. Available from: http://www.cs.utah.edu/~germain/PPS/Topics/interfaces.html [cited April 25, 2017].

3 Introducing JSON;. Available from: http://www.json.org/ [cited April 25, 2017].

4 Apple Inc . App Programming Guide for iOS; 2017. Available from: https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html [cited April 25, 2017].

5 Rosen L. Open source licensing. Prentice Hall PTR; 2004.

6 Teixeira P. Professional Node.js. John Wiley Sons, Inc; 2013.

7 Martin RC. Agile Principles, Patterns, and Practices in C. Prentice Hall; 2006.

8 Apple Inc . Cocoa Application Competencies for iOS; 2013. Available from: https://developer.apple.com/library/content/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html [cited April 25, 2017].

9 Takahashi D. The app economy could double to $101 billion by 2020; 2016. Available from: http://venturebeat.com/2016/02/10/the-app-economy-could-double-to-101b-by-2020-research-firm-says/.

10 Apple. Amount of App in Apple Store; 2016. Available from: https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/ [cited January 2017].

11 Martin J. Strategy Analytics: Subscription revenue forecast to grow from 3% to 20% of iOS app revenue by 2021. Strategy Analytics; 2015. Available from: https://www.strategyanalytics.com/strategy-analytics/news/strategy-analytics-press-releases/strategy-analytics-press-release/2015/10/29/strategy-analytics-subscription-revenue-forecast-to-grow-from-3-to-20-of-ios-app-revenue-by-2021#.WMWMShKGOis.

12 Intelliware. Next-Generation Mobile Apps – 7 Critical Success Factors; 2010. Available from: http://i-proving.com/wp-content/uploads/2010/05/White-Paper-Next-Generation-Mobile+Apps-May2010.pdf.

13 Architech Solutions. The Importance of Software Architecture;. Available from: http://static.architech.ca/wp-content/uploads/2010/06/The-Importance-of-Software-Architecture.pdf [cited April 5, 2017].

14 Apple Inc . Model-View-Controller; 2015. Available from: https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html [cited February 2017].

15 Hollemans M. Mixins and traits in Swift 2.0. Matthijs Hollemans; 2015. Available from: http://machinethink.net/blog/mixins-and-traits-in-swift-2.0/ [cited February 2017].

16 Kay A. The Early History of Smalltalk. New York: ACM; 1996.

17 Krasner GE, Pope ST. A cookbook for using the model - view controller user interface paradigm in Smalltalk - 80. 1550 Plymouth Street Mountain View; 1998.

18 Orlov B. iOS Architecture Patterns. Medium; 2015. Available from: https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52#.npqi3rbdc [cited February 3, 2017].

19 Apple Inc. Key-Value Observing Programming Guide; 2016. Available from: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html [cited April 25, 2017].

20 Microsoft. The MVVM Pattern; 2012. Available from: https://msdn.microsoft.com/en-us/library/hh848246.aspx [cited February 2017].

21 Abrahams D. Protocol-Oriented Programming in Swift. Apple; 2015. Available from: https://developer.apple.com/videos/play/wwdc2015/408/?time=483.

22 Lafore R. Object-Oriented Programming in C++, Fourth Edition. 800 East 96th St., Indianapolis, Indiana 46240 USA: Sams Publishing; 2002.

23 Apple. Protocols. Apple; 2015. Available from: https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html.

24 Apple Inc . Swift 3 - The powerful programming language that is also easy to learn.; April 1, 2017. Available from: https://developer.apple.com/swift/.

25 Apple. Extensions. Apple; 2015. Available from: https://developer.apple.com/library/prerelease/content/documentation/Swift/Conceptual/Swift_Programming_Language/Extensions.html.

26 Scala Documentation;. Available from: http://docs.scala-lang.org/tutorials/tour/traits.html [cited April 24, 2017].

27 Groovy Documentation;. Available from: http://docs.groovy-lang.org/next/html/documentation/core-traits.html [cited April 24, 2017].

28 PHP Documentation;. Available from: http://php.net/manual/en/language.oop5.traits.php [cited April 24, 2017].

29 Fisher K, Reppy J. A typed calculus of traits. 2003;Available from: http://people.cs.uchicago.edu/~jhr/papers/2004/fool-traits.pdf.

30 Hunt A, Thomas D. The Pragmatic Programmer: From Journeyman to Master. Addison Wesley; 1999.

31 Stoica M, Mircea M, Ghilic-Micu B. Software Development: Agile vs. Traditional. Informatica Economica. Romania: Bucharest University of Economic Studies, Romania; 2013.

32 Node js Foundation. Overview of Blocking vs Non-Blocking;. Available from: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/ [cited April 24, 2017].

33 Herron D. Node Web Development (2nd Edidtion). UK: Packt Publishing Ltd; 2013.

34 Holmes S. Node Web Development (2nd Edidtion). Getting MEAN with Mongo, Express, Angular, and Node. Man- ning Publications Co; 2013.

35 Sanchez R. Comparing Node.js vs PHP Performance; 2016. Available from: http://www.hostingadvice.com/blog/comparing-node-js-vs-php-performance/ [cited April 1, 2017].

36 Microsoft. Data Binding Overview; 2017. Available from: https://msdn.microsoft.com/en-us/library/ms752347(v=vs.110).aspx [cited April 25, 2017].

37 Apple Inc . iOS Human Interface Guidelines;. Available from: https://developer.apple.com/ios/human-interface-guidelines/overview/design-principles/ [cited April 1, 2017].

38 Knoernschild K. Java Design: Objects, UML, and Process; 2002.

39 Murashev N. Introduction to Protocol-Oriented MVVM;. Available from: https://news.realm.io/news/doios-natasha-murashev-protocol-oriented-mvvm/ [cited April 18, 2017].

40 Furrow A. MVVM in Swift;. Available from: http://artsy.github.io/blog/2015/09/24/mvvm-in-swift/ [cited April 13, 2017].

41 Marisi Brothers. Protocol oriented loading of resources from a network service in Swift;. Available from: http://www.marisibrothers.com/2016/07/protocol-oriented-loading-of-resources.html [cited April 18, 2017].

42 Bauer V, Heinemann L. A Structured Approach to Assess Third-Party Library Usage;Available from: http://www4.in.tum.de/~bauerv/docs/Bauer2012adequate.pdf [cited April 4, 2017].

43 Apple Inc . The Swift Programming Language. Apple Inc; 2014.

44 Tillage T. iOS 9 Tutorial Series: Protocol-Oriented Programming with UIKit;. Available from: http://www.captechconsulting.com/blogs/ios-9-tutorial-series-protocol-oriented-programming-with-uikit [cited April 14, 2017].

1 Using Traits to Create CSS-like Styling in Swift

This appendix demonstrates the progress of creating the CSS-like styling environment mentioned in section 6.3. The implementation is copied and modified from Tyler Tillage's article [44].

Firstly, a root protocol is required as seen in listing 9. Any class plans to have the CSS-like styling ability must extend this protocol.

```swift
protocol CSSStyling {
    func applyCSSStyling()
}

protocol FontType: CSSStyling {
    var name : String { get }
    var size : CGFloat { get }
    var align : NSTextAlignment { get }
}

protocol Background: CSSStyling {
    var backgroundColor : UIColor { get }
}

protocol Border: CSSStyling {
    var cornerRadius : CGFloat { get }
    var borderWidth : CGFloat { get }
    var borderColor : CGColor { get }
}
```

Listing 9: Root protocols

As also observed from listing 9, a group of styling protocols also inherit from this root. They contain general, abstract attributes and act as the roots of all other specific styling protocols. This inheritance structure is presented in listing 10.

```swift
protocol HeadingFont: FontType{}

extension HeadingFont{
    var name : String {
        return "HelveticaNeue-Bold"
    }

    var size : CGFloat {
        return 20.0
    }

    var align: NSTextAlignment {
        return NSTextAlignment.center
    }
}

protocol RoundedBorder: Border {}

extension RoundedBorder {
    var cornerRadius: CGFloat {
        return 5.0
    }

    var borderWidth: CGFloat {
        return 2.0
    }

    var borderColor: CGColor {
        return UIColor.red.cgColor
    }
}

protocol WhiteBackground: Background {}

extension WhiteBackground {
    var backgroundColor: UIColor {
        return UIColor.white
    }
}
```

Listing 10: Styling protocols

The root styling protocols let their descendants supply their own concrete attributes. As displayed in listing 10, the descendant's naming style is similar to CSS's convention. Furthermore, the attributes are enclosed within the extension scope so as to prevent outer objects from accessing and modifying the original data. These acts provide better readability for new developers as well as withdraws their concerns from the inner implementations. Ultimately, they only need to focus on the actual configuration happens in listing 11.

```swift
extension UILabel: CSSStyling {
    internal func applyCSSStyling() {
        if let s = self as? FontType {
            self.font = UIFont(name: s.name, size: s.size)
            self.textAlignment = s.align
        }

        if let s = self as? RoundedBorder {
            self.layer.borderColor = s.borderColor
            self.layer.borderWidth = s.borderWidth
            self.layer.cornerRadius = s.cornerRadius
        }

        if let s = self as? Background {
            self.backgroundColor = s.backgroundColor
        }
    }
}

class HeadingLabel : UILabel, HeadingFont, RoundedBorder,
↪  WhiteBackground {
    override init(frame: CGRect) {
        super.init(frame: frame)
        applyCSSStyling()
    }
}

class NormalTextLabel : UILabel, NormalTextFont, RedBackground{}
```

Listing 11: CSS-like styling for a label

Detailed implementation for CSS styling is injected into a UILabel's extension, as seen in listing 11. Thus, its subclass only need to conform the preferred attributes. Then, the method for styling (`applyCSSStyling` in this case) is called within the constructor and the configuration is complete. A `HeadingLabel`'s render is displayed in figure 19.

Protocol's verbose power is demonstrated clearly in listing 11. The developer does not waste time looking at the inner codes to predict the appearance of the label. Furthermore, by looking at the `NormalTextFont` declaration in listing 11, the developer expects it to have a red background, no border and a normal font face. The result image is displayed in figure

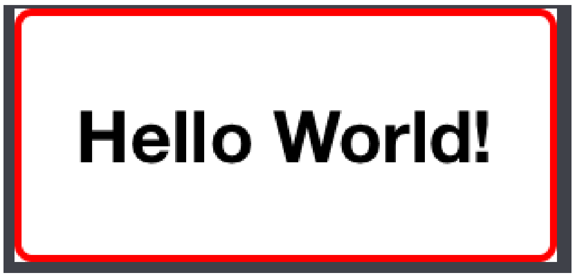


Figure 19: HeadingLabel's render

20 to verify these assumptions.

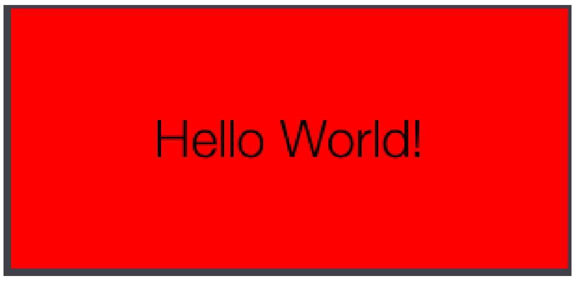


Figure 20: NormalTextLabel's render

Expectations were met and little effort was given. Not only that, this CSS-styling system could be wrapped up into a library and exported for future usage. This system is highly modifiable and requires little documentation or guidance, especially for the case of developers with web development background.