Dang Hai Phan

Benchmarking Common Architectural Patterns in iOS Development

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

04 April 2019

Metropolia

Author Title Number of Pages Date	Dang Hai Phan Benchmarking Common Architectural Patterns in iOS Development 40 pages + 3 appendices 04 April 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructors	Patrick Ausderau, Principle Lecturer

Mobile applications have exploded in popularity in the last decade. Building a mobile application can cost between a thousand dollars to hundreds of thousands of dollars. Therefore, developers should consider some major factors before developing an application whether it is application design, list of features, development time or market research. One of the most important factors is the application architecture. Good architecture can bring many benefits to the application. Some can help define separate roles and responsibilities amongst components whereas the others may result in better testability.

Therefore, the major objective of this thesis is to study and demonstrate some most common architectural patterns used in iOS development, namely Model-View-Controller (MVC), Model-View-Presenter (MVP), Model-View-ViewModel (MVVM) and View-Interactor-Presenter-Entity-Router (VIPER). The benefits and drawbacks of each pattern in terms of testability, distribution, and ease of development are also addressed. Also, considerable attention has been paid to benchmarking their performance including central processing unit (CPU) usage and random-access memory (RAM) usage.

The final product of this thesis is an iOS music application called Musicify, which was fully written in Swift. Notably, there are four versions of Musicify and each of them is implemented with a different architectural pattern. Moreover, for learning purposes, the source code of the application is made available for use and modification and can be found at https://github.com/akzuki/Musicify.

In conclusion, this thesis shows different approaches to architect an iPhone Operating System (iOS) application and their extensive impact on the quality of software development. Furthermore, it encourages developers to try and experience different architectural patterns for a better understanding of their strengths and weaknesses in order to choose the right one for an application.

Keywords	iOS, MVC, MVP, MVVM, VIPER, Testability, Distribution of Responsibilities, Ease of Use

Metropolia

Contents

List of Abbreviations

1 Introduction 1

2 Theoretical background 3

2.1 Architectural patterns 3
2.2 Common architectural patterns in iOS development 5
2.2.1 Traditional MVC and Apple's MVC 5
2.2.2 MVP 7
2.2.3 MVVM 9
2.2.4 VIPER 10
2.3 Relevant technologies 11
2.3.1 XCode 11
2.3.2 Swift programming language and Cocoapods 12

3 Implementation 15

3.1 Introduction to Musicify application 15
3.2 Network layer 16
3.3 Model 20
3.4 Apple's MVC 22
3.5 MVP 23
3.6 MVVM 26
3.7 VIPER 28

4 Pattern analysis 32

4.1 Distribution of responsibilities 32
4.2 Testability 34
4.3 Ease of use and performance 35

5 Conclusion 37

References 38

Appendices
Appendix 1. Musicify's Github repository

Metropolia

Appendix 2. Full implementation of MusicifyAPI network layer using Moya

Appendix 3. Project file structure

Metropolia

List of Abbreviations

API	Application Programming Interface
BDD	Behavior-Driven Development
CPU	Central Processing Unit
GUI	Graphical User Interfaces
HTTP	HyperText Transfer Protocol
HTTPs	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
iOS	iPhone Operating System
JSON	Javascript Object Notation
KVO	Key-Value-Observing
MacOS	Macintosh Operating Systems
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
RAM	Random Access Memory
SDKs	Software Development Kits
UI	User Interface
UIKit	User Interface Kit

VIPER	View-Interactor-Presenter-Entity-Router
XAML	Extensible Application Markup Language

1 Introduction

The global mobile application market is growing at an ever-increasing rate and it shows no signs of stopping. According to a retrospective report conducted in 2017 by App Annie, the leading global provider of app market data, the total gross consumer spend on app stores has doubled from slightly over $40 billion in 2015 to $86 billion in 2017, marking 105% growth over a two-year period [1]. It is also worth mentioning that there has been a significant increase in the number of applications in the app stores compared to the last few years. As illustrated in figure 1, as of the third quarter in 2018, there were over 2 million mobile applications available to download in Google Play Store and 2 million applications in the Apple App Store, two leading app stores in the world. The number of applications in other app stores such as Windows Store, Amazon App Store, and BlackBerry World varied from over 234 thousand to 669 thousand, making up a total of 1.3 million applications. [2]

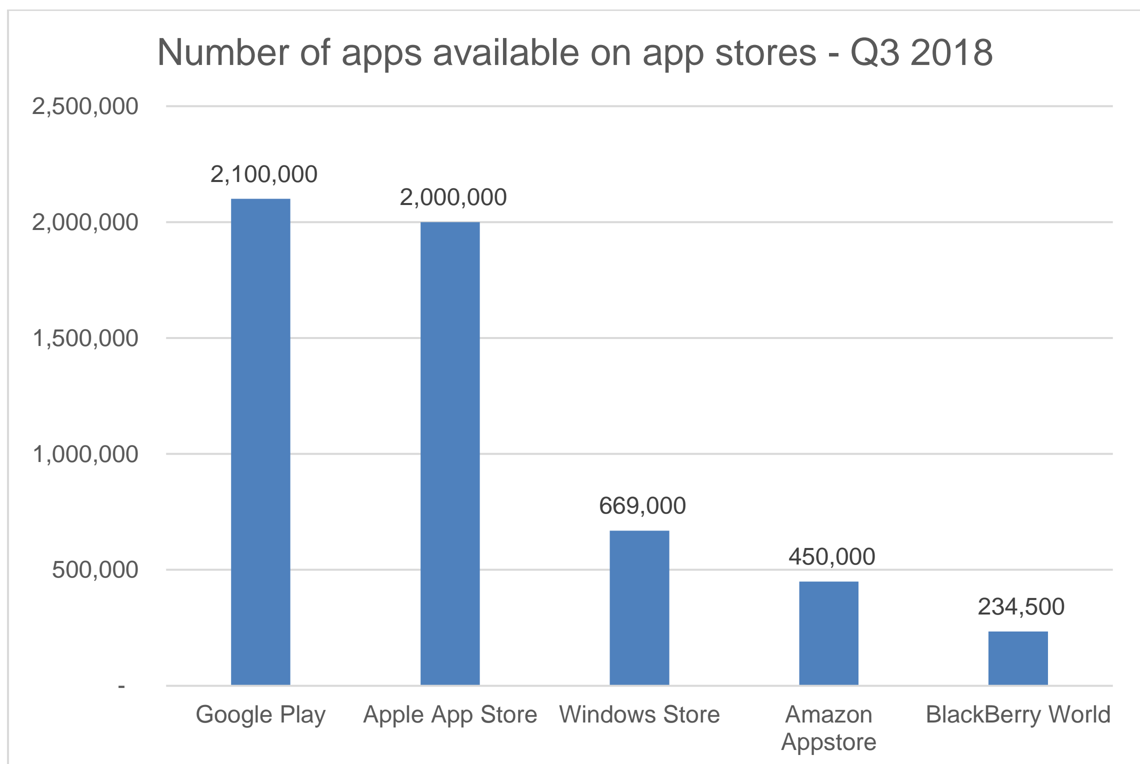

Figure 1. The number of applications available on app stores as of the third quarter in 2018. Summarized from Statista (2018) [2].

However, building a functional and well-structured application remains a very challenging task. There is a wide variety of factors that contribute to the successful development of an application. Having a software architecture is certainly one amongst them. [3, p.3] As stated by Mark Richards in his book Software Architecture Patterns (2015) [4], software architecture helps characterize the application and define its behavior. Some architectural patterns focus on high scalability whereas others can result in high agility. [4, p.5] Despite their benefits, it is commonly known that architectural patterns have not been taken into consideration by developers in the early phase of development. Due to the nature of software development where changes are unavoidable, an application tends to grow in size and functionality from time to time. Thus, developing an application without a proper architectural pattern can be burdensome and quickly leads to unorganized, unmaintainable source-code and tightly-coupled modules. As a result, introducing new changes in such an application without a clear vision or directions can become troublesome. [4, p.5]

In essence, the primary goal of this study is to demonstrate some most commonly used architectural patterns in iOS development such as MVC, MVP, MVVM, and VIPER. Furthermore, these patterns are put into comparison in terms of distribution of responsibilities, testability, and ease of use, thus giving the practitioners a better understanding of their strengths and weaknesses. The performance variations between these patterns are also addressed in this study.

2 Theoretical background

2.1 Architectural patterns

A software architectural pattern is a well-proven and reusable solution to recurring problems in software systems. In other words, a pattern is usually created in a situation in which certain problems are solved by many developers using a similar approach and the solution is generally acknowledged and well-documented. Thus, it becomes a pattern. [5, p.36]

An architectural pattern builds a connection between a context, a problem, and a solution. More specifically, a context is a commonly occurring situation which leads to a problem. A problem occurs in the given context. It is notable that the problem should be generalized appropriately. Therefore, the problem and its variants are described in the pattern description. Finally, a solution is an effective resolution to the given problem. For this reason, a solution aims to illustrate the architectural structure which provides a strategy to solve the problem. There are numerous factors that determine a pattern solution. These factors include a set of element types such as data repositories, processes, and objects, a set of interaction mechanisms namely events, message bus, among others, and a layout of the components. Finally, a set of semantic constraints in which the topology, the element behavior, and the interaction mechanisms are covered. [3, p.204]

To give a better understanding about the architectural pattern and the relationships between a context, a problem and a solution in such pattern, an example of architectural pattern named client-server pattern which can be observed in various distributed systems is described in figure 2.

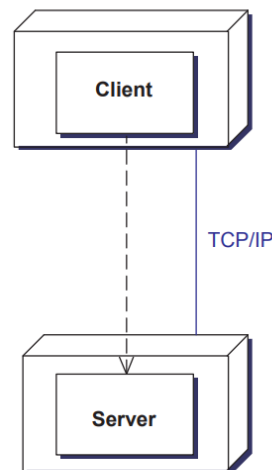

Figure 2. Client-server pattern. Copied from Open Universiteit (2014) [5, p.39].

As seen in figure 2, the client-server pattern consists of two components namely the clients and the servers. Communication between these components is carried out through protocols and messages shared amongst them to handle the system's work. [3, p.18] In this architectural pattern, service requests are sent from the clients to the servers in which each request is processed. Consequently, needed services are delivered back to the clients. [5]

In this case of the client-server pattern, the context is a situation where there are a great number of distributed clients which need accessing shared resources and services. Hence, the need for access control and quality control for those resources arises. The problem is that in order to promote modifiability and reuse, common services and the need for adjusting these services in certain locations have been factored out. However, further improvements in terms of scalability and availability are expected. This could be achieved by centralizing the control of these resources and services whereas making the resources distributed across different physical servers. Finally, the proposed solution for this problem is known as when the clients send requests for services to the servers. Once the request has been processed by the servers, a set of services are sent back to the clients as a return. [3, pp.217-219]

Finally, it is worth mentioning that in real-world applications, multiple patterns of architecture can be applied at once in a single system. This combination of multiple patterns can be observed in many complex systems. A web-based system, for example, might

exploit the client-server pattern, however, inside this pattern, it might utilize caches, fire-walls, proxies, MVC and more. Furthermore, within each of them, more patterns and tactics might be applied, making each module optimized with the best architecture. [3, p.204]

2.2 Common architectural patterns in iOS development

There is a wide range of architectural patterns applied in iOS development such as MVC, MVP, The Elm Architecture, Redux Architecture, to name a few. However, this study focuses on the most commonly used pattern for architecture amongst iOS developers.

2.2.1 Traditional MVC and Apple's MVC

MVC, which was initially introduced by Trygve Reenskaug in the Smalltalk-76 in the 1970s, is a widely known architectural pattern used primarily in creating graphical user interfaces (GUIs). The core idea of MVC is to separate the application's concerns into different roles. For this reason, MVC has exploded in popularity amongst developers since its first appearance. [6] The relationships between MVC components are described in figure 3.

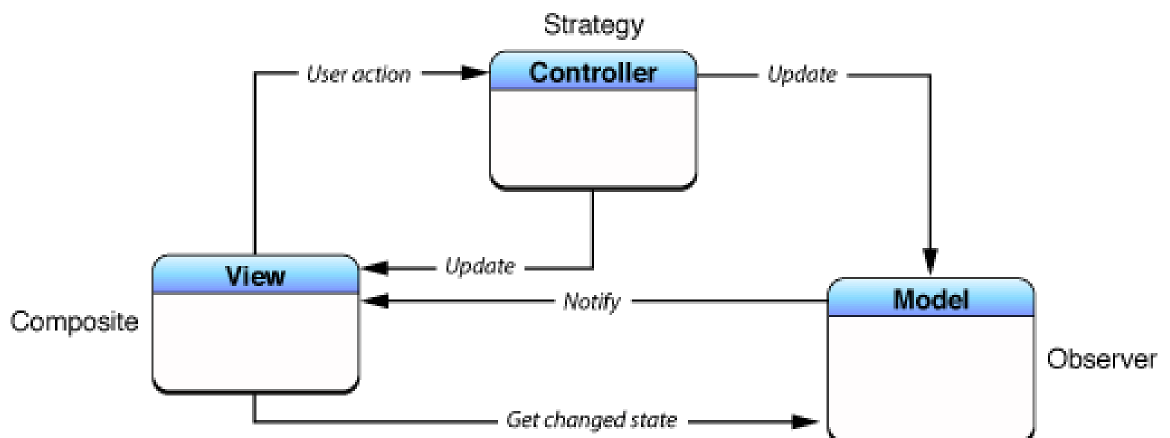

Figure 3. The traditional MVC pattern. Copied from Apple Inc. (2012) [7].

As illustrated in figure 3, MVC divides application's module into three separate roles: Model, View, and Controller which are separated from each other by abstract bounda-

Metropolia

ries. The Model contains business logic and defines the rules of application. [7] For example, the Model can store, encapsulate and abstract data in an application. This can be as simple as a string to store the user's name, or it can be a User object that saves detailed information of the user including first name, last name, age and so forth. Some tasks which are related to data management such as data retrieval, data validation, and data persistence also belong to the Model. Notably, it is recommended by Apple that the Model should be implemented in such a way that all application's essential data is encapsulated inside it [7].

On the other hand, the View takes responsibility for the presentation logic of the application. Thus, its primary task is to display the data to the users and handle user interactions. As observed from figure 3, the View is notified whenever there are any changes that take place inside the Model. As a result, it retrieves the latest application's state from the Model and redraws its content. [7]

Finally, the Controller acts as a mediator between the View and the Model. It receives user interaction events from the View and updates the Model accordingly. In addition, the Controller can also manage the life cycles of other components. [7]

The traditional MVC architectural pattern can be adapted in many instances varied from web applications, desktop applications to mobile applications [6]. However, as the basic patterns used in traditional MVC differs from ones used in Apple's native object-oriented Application Programming Interface (API), a new variant of MVC was introduced by Apple [7].

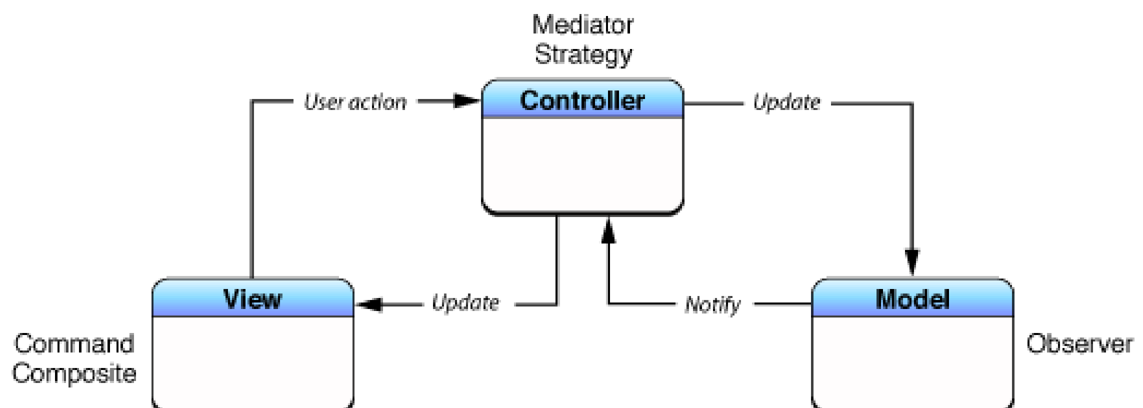

Figure 4. Apple's MVC architectural pattern. Copied from Apple Inc. (2012) [7].

Metropolia

As depicted by figure 4, Apple's version of MVC keeps the core idea of separation of concerns from the traditional MVC with three components as Model, View, and Controller. Correspondingly, the application logic still resides inside the Model whereas the presentation logic belongs to the View. However, there is a noticeable difference in Apple's MVC compared to the traditional one. Unlike the traditional MVC where the View gets notified by the Model and can retrieve data from it, the Model no longer communicates directly to the View and vice versa. In fact, as suggested by Apple, the View and the Model should not know about each other in an ideal MVC application [7]. Instead, the Model informs the Controller of any changes in the application state. The Controller, in turn, receives such notifications and asks the View to display with new data. At the same time, user actions are passed from the View to the Controller which then updates the model consecutively.

By this design, Apple's MVC gains some advantages over traditional MVC. Most notably, the reusability of the View and the Model has been enhanced as they are isolated from each other. Furthermore, parallel development is also possible. Thus, developers can update the business logic inside the Model without impacting the presentation logic that belongs to the View and vice versa. [7]

2.2.2 MVP

First introduced at Taligent in the early 1990s, MVP is one of the first alternatives to MVC pattern with the aim to provide a powerful yet straight-forward architecture solution for building applications and, at the same time, to improve the separation of concerns [8, p.1]. The communications between MVP's components are described in figure 5.

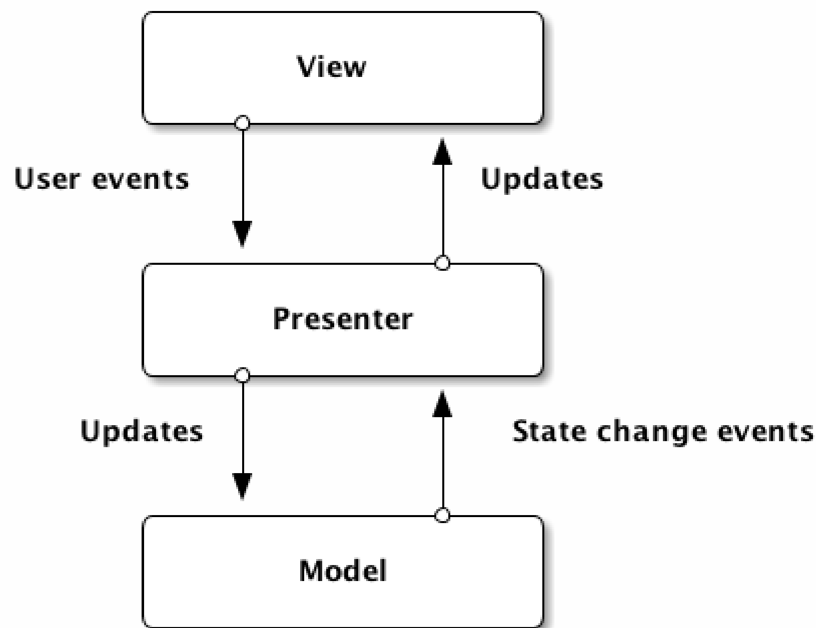

Figure 5. MVP architectural pattern. Copied from Almiray (2015) [9].

As a derivation of MVC, MVP shares some similarities with its origin. It can be observed from figure 5 that there are three key components involved in MVP pattern namely Model, View, and Presenter. Similarly, the Model is still responsible for the domain data access and manipulation. The View, on the other hand, contains the presentation logic. Therefore, its primary task remains displaying data to the users and passing user events to other components. [10] However, MVP introduces a new component: Presenter to replace the Controller in MVC. As the middle-man, the Presenter is responsible for keeping the application synchronized. For this reason, it handles user interactions from the View and updates the model accordingly. And, at the same time, the Presenter listens to state change events from the Model, converts the data into a user-interface-friendly format and then updates the View when necessary.

Although it seems that relationships between MVP components look almost identical to Apple's MVC, MVP has its own characteristics. Most notably, the Presenter is completely user interface kit (UIKit) independent compared to the Controller [11]. In other words, there is no existence of User Interface (UI) elements such as buttons, labels and text fields inside the Presenter. Moreover, handling view life cycles is no longer the responsibility of the Presenter [11].

2.2.3 MVVM

MVVM made its first appearance in the early 2000s at Microsoft to provide a simple solution to design and develop Extensible Application Markup Language (XAML) application platforms such as Silverlight [12]. Later, it has gained popularity in the iOS community for building responsive application [13]. Figure 6 illustrates the different layers in MVVM and its data flow.

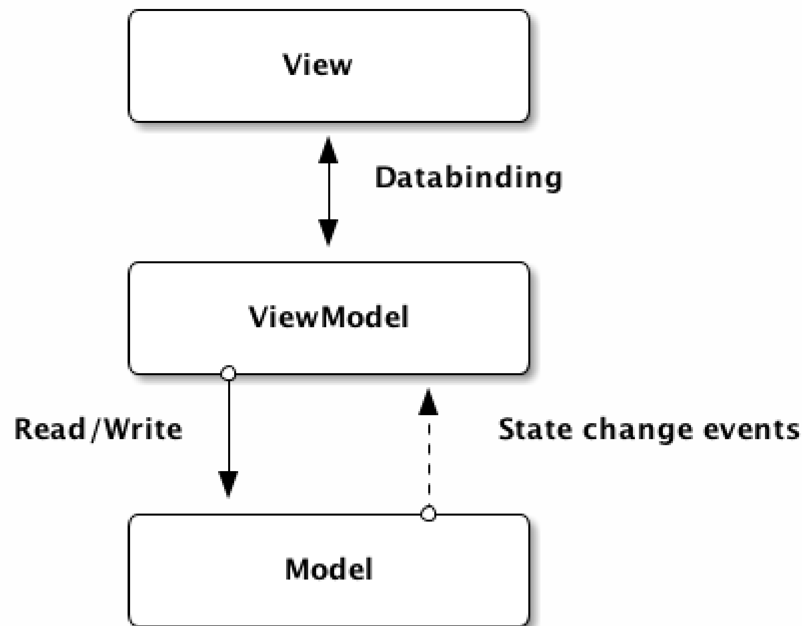

Figure 6. MVVM architectural pattern. Copied from Almiray (2015) [9].

MVVM is composed of three components: Model, View, and ViewModel, as observed from figure 6. Similar to other Model-View-* patterns that have been mentioned in this study, the Model in MVVM contains business logic of the application, meaning that data persistence, data manipulation, networking logic and such often reside here [11]. The View still takes responsibility for rendering UI components and displaying data to the user. Communication between the View and the Model is carried out through a new component called ViewModel. Likewise, it acts as a middle-man that keeps the data flow running smoothly. The ViewModel is capable of updating the Model and listening to state change events whenever new changes occur in the data. The core idea behind MVVM pattern is that the View is backed by the ViewModel which reformats the data for it to

Metropolia

display. In addition, the ViewModel is also UIKit independent like the Presenter in MVP pattern. [12]

However, there is a noticeable difference in the way the View and the ViewModel communicate. That is, the ViewModel is owned by the View but the other way around may not apply, meaning that it does not necessarily hold a reference to the View. Instead, the communication between them is often carried out through data bindings. [11] More specifically, UI events triggered inside the View are observed and populated into the ViewModel. At the same time, the View can be notified by the ViewModel about changes in application data. This is also known as two-way binding. As a result, the application tends to become more responsive as the View gets updated constantly to reflect the latest application state.

2.2.4 VIPER

As derived from the Clean Architecture by Robert C. Martin [14], VIPER does not fall into the Model-View-* category. However, it still aims to improve the separation of concerns of an application by dividing into distinct layers of responsibilities [15]. These layers are illustrated in figure 7.

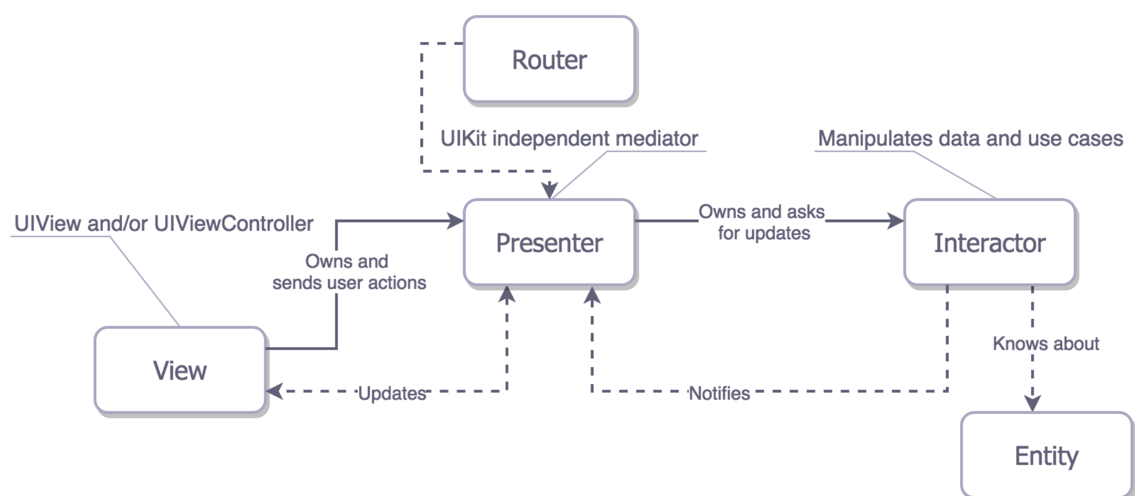

Figure 7. VIPER architectural pattern. Copied from Orlov (2015) [11].

As observed from figure 7, VIPER has five separate layers that are View, Interactor, Presenter, Entity, and Router. Firstly, the main responsibility of the VIPER's View remains unchanged compared to the View in MVP. That is data visualization and user

interaction handling. More specifically, it delegates any user-triggered events to the Presenter and waits for the content that it should display next. [15] Secondly, the Presenter receives and acts upon such events. Noticeably, it does not own the Model like the Presenter in MVP. Instead, the Presenter holds a reference to a new component called Interactor and asks for data updates from it.

The Interactor encapsulates the business logic and usually represents the use cases of an application. For example, in a to-do application, a use case can be fetching a list of to-do items stored in a remote database. In that case, the Interactor often initiates fetching action through a networking dependency and then informs the Presenter about the obtained result. Additionally, the Interactor knows about and manipulates the Entity, which is a plain data object. It is important to note that data access logic does not reside in the Entity. In fact, such responsibility belongs to the Interactor. [15]

Finally, VIPER introduces a new component called Router that is specifically for handling navigation logic between VIPER modules. The Router receives events from the Presenter telling which screen should be displayed to the user next. Therefore, it configures a new `View` or `ViewController` object and then performs the transition to the new screen. [15]

2.3 Relevant technologies

2.3.1 XCode

First released in 2003, XCode is Apple’s integrated development environment (IDE) for Macintosh operating systems (macOS). It provides a wide variety of tools for developing applications for macOS, iOS and other operating systems from Apple, such as Interface Builder, Source Editor, Continuous Integration, XCTest Framework, and iOS Simulators. [16]

Interface Builder is a visual tool to help developers design and test user interface inside XCode without the need to write down a single line of code. This can be achieved by just simply dragging and dropping labels, text fields, buttons, switches, and other elements onto the design canvas. Auto Layout, whose core idea is to provide each element in the interface a constraint to define the way it behaves to other elements, is also built into Interface Builder. [17] A constraint placed on a view acts as a guideline to help Auto

Metropolia

Layout calculate the size and position of such view in the view hierarchy. With proper use of constraints, Auto Layout allows developers to create user interfaces that are responsive to different screen sizes and orientations. [18]

XCTest Framework is an extremely helpful testing framework integrated into XCode for automated testing. It has many feature-rich software testing capabilities which enable the users to create and execute basic unit tests, performance tests, and UI tests for XCode projects. [19] Moreover, XCTest is fully compatible with Objective-C and Swift and its tests can be executed in a simulator or a real physical device [20].

iOS Simulators are introduced as a major part of XCode toolset. They are used not only to quickly model applications but also to see the aftereffects of changes rapidly. Therefore, iOS simulators can be used to run tests and debug errors in the early phase of development. [21] However, it is recommended by Apple that applications should be tested on real physical devices due to differences in hardware and API between simulators and real devices [22].

Instruments, which is shipped with XCode, is a powerful tool to analyze an application's performance and identify potential issues early in the development cycle. Instruments contain a list of specialized tools, namely Allocations, Visual Memory Debugger, Time Profiler. Each of them helps developers analyze different parts of an application. For example, memory problems such as leaks, abandoned memory can be detected by using the Allocations tool. On the other hand, CPU usage and power consumption can be measured with the help of Instruments' profiling tool. Additionally, the data is also gathered as Instruments profiles and the outcome is displayed in detail for analysis afterward, thus giving a better understanding of an application's behavior and performance. [23]

In addition, XCode supports a great number of programming languages, namely C, C++, Objective-C, Objective-C++, Java, AppleScript, Python, Ruby, ResEdit, and Swift. As of the time of writing this study, the latest version of XCode is 10.1 which is available to download on the Mac App Store. [16]

2.3.2 Swift programming language and Cocoapods

Swift is a general-purpose and multi-paradigm programming language developed by Apple. Along with Objective-C, Swift is also used to write applications for macOS, iOS and

Metropolia

other operating systems from Apple. [24] First introduced at Apple's 2014 Worldwide Developers Conference, Swift 1.0 has quickly gained popularity amongst mobile developers. Some of Swift's features are access control, optionals and chaining, value types, memory management, and protocol-oriented programming [25]. Thanks to those modern features, it made to the first place in the Most Loved Programming Language list in a survey conducted by Stackoverflow in 2015 [26].

Additionally, one of the new features released in Swift 4.0 is Codable. Its intent is to provide an easy way to encode and decode data formats such as JSON to native Swift objects and vice versa, thus making the conversion between these formats incredibly smooth. [27] Prior to Swift 4.0, mapping JSON to native Swift objects is usually done by using the Foundation framework's `JSONSerialization` or with the help of third-party libraries such as ObjectMapper[1], SwiftyJSON[2], and Arrow[3].

Moreover, Swift has been open-source since 2015, thus allowing more developers to contribute their fixes and enhancements to the language [25]. Later major versions of Swift are 2.0, 3.0 and 4.0 which were released on 21 September 2015, 13 September 2016 and 19 September 2017, respectively.

Each programming language has its own dependency management tools and Swift is definitely no exception. Cocoapods is a dependency manager for Swift and Objective-C Cocoa projects. Its primary goal is to allow users to define their applications' dependencies inside a file called Podfile and then manage their versions easily over time and in different development environments. Since the first public release in 2011, Cocoapods has been used in over 3 million apps and its number of available libraries has reached to over 55 thousand libraries. [28] Cocoapods can be installed on macOS by running the command `sudo gem install cocoapods` in the terminal. Once the installation is completed, users can create a text file named Podfile and add the desired dependencies into the list.

```
1 # Uncomment the next line to define a global platform for your pro-
ject
2 # platform :ios, '9.0'
3
4 target 'Musicify' do
```

[1] https://github.com/tristanhimmelman/ObjectMapper
[2] https://github.com/SwiftyJSON/SwiftyJSON
[3] https://github.com/freshOS/Arrow

Metropolia

```
5   # Comment the next line if you're not using Swift and don't want
to use dynamic frameworks
6   use_frameworks!
7
8   # Pods for Musicify
9   pod 'Moya', '12.0'
10  pod 'Kingfisher', '5.0.1'
11
12  target 'MusicifyTests' do
13    inherit! :search_paths
14    # Pods for testing
15  end
16
17  target 'MusicifyUITests' do
18    inherit! :search_paths
19    # Pods for testing
20  end
21
22 end
```

Listing 1. An example of Podfile.

A Podfile, which can be created with smart defaults by running `pod init` command in the terminal, contains the dependencies of the targets of one or more XCode projects [28]. As shown in listing 1, there are two libraries listed in the Podfile namely Moya and Kingfisher with version 12.0 and 5.0.1, respectively. By running `pod install` command, Cocoapods will install these libraries and generates the XCode workspace `.xcworkspace`, which can be used to build the projects with installed dependencies.

Metropolia

3 Implementation

3.1 Introduction to Musicify application

In order to demonstrate some architectural patterns, an iOS application called Musicify, which is derived from a possible real-world scenario, has been carried out. Musicify is a music application with the aim to provide its users latest songs in the billboard along with their information. Musicify fetches the song data from Myjson API[4] which then returns the latest songs in JavaScript Object Notation (JSON) format. As depicted in figure 8, the main view shows the most recent songs in a list view. Each song contains the name of the song, the name of the artist, the year of release and the thumbnail image. The app allows users to select a song and as a result, they will be redirected to Azlyrics[5] page where more information about the song as well as its lyrics is presented.

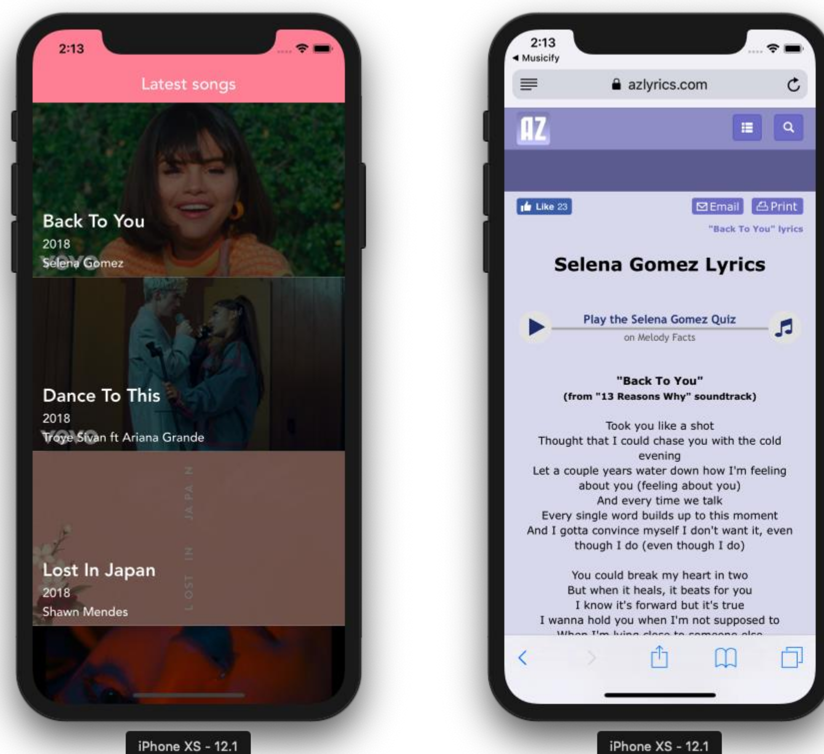

Figure 8. Musicify application: (left) main view and (right) lyrics view

4 http://myjson.com/api
5 https://www.azlyrics.com/

Metropolia

Musicify is not a complex application. It is important to note that mobile applications come in all shapes and sizes with different target audiences, feature sets, and use cases. Some applications might need to fetch data from external resources and then persist those data for later use whereas the others might have features that require location-based tracking services, voice services and so forth. Thus, due to the huge variety in terms of application functionality, it is a challenging, if not impossible, task to conduct an application which has all the features offered by the iOS operating system for demonstration purpose. Musicify, on the other hand, focuses on some common basic tasks in an iOS application such as networking, data processing, displaying data into a list view (`UITableView`[6]/`UICollectionView`[7]). Finally, unit testing is also covered in this application and as a result, some test cases will be carried out in section 3.3, 3.5, 3.6 and 3.7 of this study.

In addition, different architectural patterns are implemented and benchmarked using a MacBook Pro 2017 computer. Its technical specifications are MacBook Pro 2017 with 3.1 GHz Intel Core i5, 16GB 2133 MHz LPDDR3 memory, Intel Iris Plus Graphics 650 1536 MB and macOS Mojave 10.14 operating system. In addition, the application is written in Swift 4.2.1 and the IDE used for development is XCode 10.1, which includes Swift 4.2.1 and Software Development Kits (SDKs) for iOS 12.1. Finally, performance analysis is carried out with the help of iPhone 8 device.

3.2 Network layer

In recent years, most mobile applications require networking to connect to external services and data resources, meaning that they may interact with the backend, web services, or APIs [29]. Some of which could be provided by Google, Amazon or third-parties whereas others could be developed internally. This client-server architecture is beneficial for many applications. Not only it helps improve data sharing by allowing the multiple authorized clients to access the data on a server, but also it results in shared resources amongst different platforms.

However, as pointed out by Apple in their networking documentation, networks are unreliable by nature. For instance, in order to send the user’s data to an Internet server,

---

[6] https://developer.apple.com/documentation/uikit/uitableview
[7] https://developer.apple.com/documentation/uikit/uicollectionview

the network data passes through one to many physical interconnects whose speed can vary from 300 bits per second to almost 40 billion bits per second, making a drastic change in the speed of the user's connection. Thus, developing a good networking solution for iOS apps is a complex task. Fortunately, as suggested by Apple, there are some ways to improve the ability to adapt to changing network conditions of an iOS application. Some of which include avoiding timeouts whenever possible, failure handling, choosing appropriate APIs, allowing the user to cancel time-consuming transactions and so forth. [29]

Additionally, developers also need to take responsibility for secure communication. More specifically, all data sent by the user should be treated as confidential, thus being protected accordingly. This can be achieved by, for example, encrypting such data during transit and making sure that it is sent to trusted people or servers. Moreover, any incoming data from an untrusted source should be inspected carefully as it may be a malicious attack. [29]

As an iOS developer, one will need to be prepared to build an application with the capability to perform certain networking tasks. Some of which include performing Hypertext Transfer Protocol (HTTP) or Hypertext Transfer Protocol Secure (HTTPSequests like POST, GET, PUT, DELETE requests, listening for incoming connections and establishing a connection to a remote host. These tasks can be accomplished by using Apple-provided networking APIs. For instance, `NKAssetDownload`[8] API can be utilized to download content in the background, which is beneficial for Newsstand applications whereas `NSURLSession`[9] class, `NSURLConnection`[10] class, and related classes can be used to perform HTTP/HTTPS requests.

In Musicify application, as mentioned in section 3.1, the application sends a GET request to Myjson API endpoint to retrieve the lists of latest songs in JSON format. There are numerous ways to accomplish this task: using `NSURLSession` class, `NSURLConnection` class, and related classes or using third-party networking libraries such as Alamofire[11], AFNetworking[12], FSNetworking[13], and Moya. Out of the presented solutions, Moya is

8 https://developer.apple.com/documentation/newsstandkit/nkassetdownload
9 https://developer.apple.com/documentation/foundation/nsurlsession
10 https://developer.apple.com/documentation/foundation/nsurlconnection
11 https://github.com/Alamofire/Alamofire
12 https://github.com/AFNetworking/AFNetworking
13 https://github.com/foursquare/FSNetworking

Metropolia

chosen to handle networking requests in Musicify. Additionally, this library can be installed via Cocoapods.

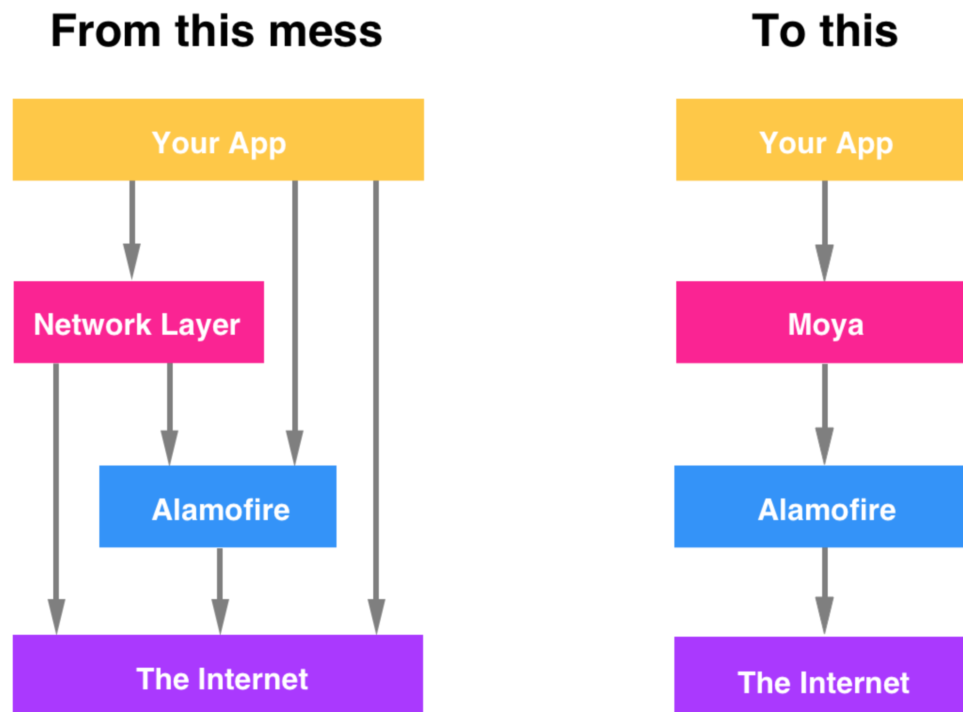

Figure 9. Moya: network abstraction layer. Copied from Moya [30].

Inspired by the concept of encapsulating network requests in a type-safe way, Moya provides users with abstraction to make network calls without directly communicating with Alamofire, as seen in figure 9, while keeps the network requests properly configured using Swift's `enum` (Enumerations[14]). This brings many benefits to the application. Firstly, Moya ensures that all API endpoint accesses are correct due to the use of type-safe `enum`. Secondly, as all relevant API service logic is encapsulated in one place, it makes structuring network services and requests more approachable, making the usage of

[14] https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html

those API endpoints clearer and much more well-defined. Last but not least, implementing Moya in an application can result in better testability since it allows developers to use test stubs[15] for network calls in unit testing. [30]

```
1    enum MusicifyAPI {
2        case latestSongs
3    }
4
5    extension MusicifyAPI: TargetType {
6        var baseURL: URL { return URL(string:
"https://api.myjson.com")! }
7
8        var path: String {
9            switch self {
10           case .latestSongs:
11               return "/latestSongs"
12           }
13       }
14
15       var method: Moya.Method {
16           switch self {
17           case .latestSongs:
18               return .get
19           }
20       }
21
22       var task: Task { // return the task for endpoint }
23
24       var headers: [String : String]? { return nil }
25
26       var sampleData: Data {
27           switch self {
28           case .latestSongs:
29               return (
30                   """
31                   [
32                       {
33                           "id": 1,
34                           "name": "Back To You",
35                           ...
36                       }
37                       // Other sample data
38                   ]
39                   """.data(using: .utf8)!
40               )
41           }
42       }
43   }
```

Listing 2. Network layer in Musicify application using Moya[16].

[15] A stub is a block of code that can simulate the behavior of existing code or provide ready-made answers to calls during the test [31].
[16] Full implementation of MusicifyAPI is listed in Appendix 2.

As observed from listing 2, `MusicifyAPI` is defined as an `enum` in which each API endpoint is a case of the `enum`. In this example, the `enum` contains one case `.latestSongs` since the application uses only one API endpoint from Myjson API to get the latest songs. In addition, `MusicifyAPI` needs to conform to the `TargetType` protocol from Moya. As a result, several methods need to be implemented. Those methods include `baseURL`, `path`, `method`, `task` and `headers` which will be used to eventually build the correct `Endpoint` object. Finally, `sampleData` is implemented in order to provide stubs of the API for testing. This is useful when doing unit tests as Moya can return this response for network calls without sending actual requests to Myjson API. Some examples of unit tests with Moya will be brought up in section 3.5, 3.6 and 3.7.

3.3 Model

It is common that the model layer often contains model objects and coordinating objects such as networking services that send and receive data from external APIs or objects that persist data on disk. In Musicify application, each song is a model object and is implemented as a `struct`[17].

```
1    import Foundation
2
3    struct Song: Codable {
4        let id: Int
5        let name: String
6        let artist: String
7        let releaseYear: String
8        let thumbnailURL: URL
9        let lyricsURL: URL
10   }
```

Listing 3. Song model implementation

The `Song` model contains six properties namely `id`, `name`, `artist`, `releaseYear`, `thumbnailURL`, and `lyricsURL`, as seen in listing 3. Apart from `name`, `artist`, and `releaseYear` which are intelligible, the `id` is an Int value which is used to uniquely identify an object. Therefore, two different songs will have two different identifiers. In addition, `thumbnailURL` is used to load the thumbnail image of a song whereas `lyricsURL` is used to redirect

[17] https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html

users to lyrics page as illustrated in figure 8. Notably, `Song` object conforms Swift's `Codable` protocol which is a type alias for the `Encodable` and `Decodable` protocol.

```
1    func testParsingJSONReturnsCorrectSong() {
2            // Given
3            let actualSongJSON = """
4                {
5                    "id": 1,
6                    "name": "Back To You",
7                    "artist": "Selena Gomez",
8                    "releaseYear": "2018",
9                    "thumbnailURL": "https://i.ytimg.com/vi/VY1eFxgRR-
k/maxresdefault.jpg",
10                   "lyricsURL": "https://www.azlyrics.com/lyr-
ics/selenagomez/backtoyou.html"
11               }
12           """
13
14           // When
15           guard let actualSong = try? JSONDecoder().de-
code(Song.self, from: actualSongJSON.data(using: .utf8)!) else {
16               return XCTFail("Parse JSON failed")
17           }
18
19           // Then
20           let expectedSong = Song(
21                                   id: 1,
22                                   name: "Back To You",
23                                   artist: "Selena Gomez",
24                                   releaseYear: "2018",
25                                   thumbnailURL: URL(string:
"https://i.ytimg.com/vi/VY1eFxgRR-k/maxresdefault.jpg")!,
26                                   lyricsURL: URL(string:
"https://www.azlyrics.com/lyrics/selenagomez/backtoyou.html")!
27                              )
28
29           XCTAssertEqual(actualSong, expectedSong)
30       }
```

Listing 4. Song model test case

The listing 4 demonstrates a test case for the `Song` model to assert that JSON parsing with `Codable` protocol works as expected. Firstly, a JSON object for a song is given as input parameters to the method under test. Secondly, this object and the `Song` type `Song.self` are passed to the decode function of `JSONDecoder` instance which then returns a `Song` object. Finally, the expected song object is given and compared with the actual result. In order for the test to pass, these two objects must be identical. Otherwise, if there is a mismatch between the results, the test will fail.

3.4 Apple's MVC

In Apple's MVC pattern, the `LatestSongsViewController` class, also known as the Controller in this module, mediates between the View and the Model. This is typically carried out by the delegation pattern[18]. A diagram for `LatestSongsViewController` class is illustrated in figure 10.

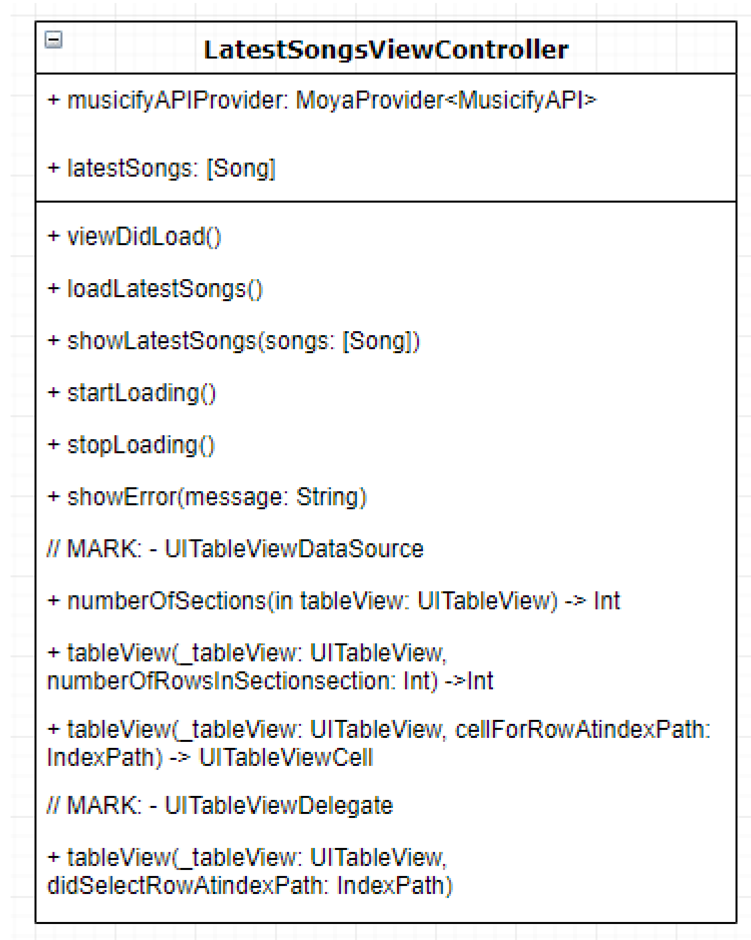

Figure 10. LatestSongsViewController implementation in MVC

Once the `LatestSongsViewController` class has loaded its view hierarchy into the memory, the `loadLatestSongs` function is called. Consequently, a GET request is sent to Myjson API to get the latest songs. This is conducted with the help of Moya library. At the same time, the application calls the `startLoading` function to display a loading icon

---

[18] https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html

indicating that fetching song data is in process. Once the song data is retrieved, mapping JSON objects to Swift objects is carried out. Finally, the controller updates the view to present the latest songs to the users in a `UITableView`. It is important to note that if there is any error during the process, the application will be able to catch the error and display an alert to end users by calling the `showError` function.

As can be seen from figure 10, the `LatestSongsViewController` class has too many responsibilities such as sending network requests, handling network response, mapping JSON to Swift objects, updating the view based on changes, just to name a few. Since `LatestSongsViewController` class is the delegate (`UITableViewDelegate`[19]) and data source (`UITableViewDataSource`[20]) of the table view, the logic for data control and behavior based on different events of the table view also resides in this class. Furthermore, `LatestSongsViewController` class has some methods to handle view lifecycles such as `viewDidLoad`, `viewWillAppear`, and `viewDidAppear`, making it tightly coupled to the UI environment. For these reasons, writing unit tests for this type of `ViewController` can be found difficult to conduct.

3.5 MVP

In terms of MVP, the `UIViewController` class is consolidated as a View component. In the case of `LatestSongsViewController` class, the main purpose of the view is to display a list of latest songs and to allow users to select a specific song to retrieve further information about that song. Also, as the app fetches the data from an API, a loading icon is needed to indicate that the request is still processing. In addition, the view should prompt a message to the users in case the request fails for some reasons. Such presentation logic can be abstracted by defining a protocol for the view called `LatestSongsView`.

---

[19] https://developer.apple.com/documentation/uikit/uitableviewdelegate
[20] https://developer.apple.com/documentation/uikit/uitableviewdatasource

```
1   protocol LatestSongsView: class {
2       func startLoading()
3       func stopLoading()
4       func showLatestSongs(songs: [Song])
5       func showError(message: String)
6   }
7
8   // MARK: - LatestSongsView
9   extension LatestSongsViewController: LatestSongsView {
10      func startLoading() {
11          // implementation for showing loading icon
12      }
13
14      func stopLoading() {
15          // implementation for hiding loading icon
16      }
17
18      func showLatestSongs(songs: [Song]) {
19          // implementation for showing latest songs in a table view
20      }
21
22      func showError(message: String) {
23          // implementation for showing error
24      }
25  }
```

Listing 5. LatestSongsView protocol implementation

As seen in listing 5, `LatestSongsView` protocol is defined with four instance method requirements: `startLoading`, `stopLoading`, `showLatestSongs`, and `showError`. This protocol does not make any assumptions about how to handle the loading state, how to display the latest songs, and how to prompt a message to the users when an error occurs. In fact, `LatestSongsView` protocol is adopted and conformed by `LatestSongsViewController` class, which then provides its own implementations for handling such presentation logic.

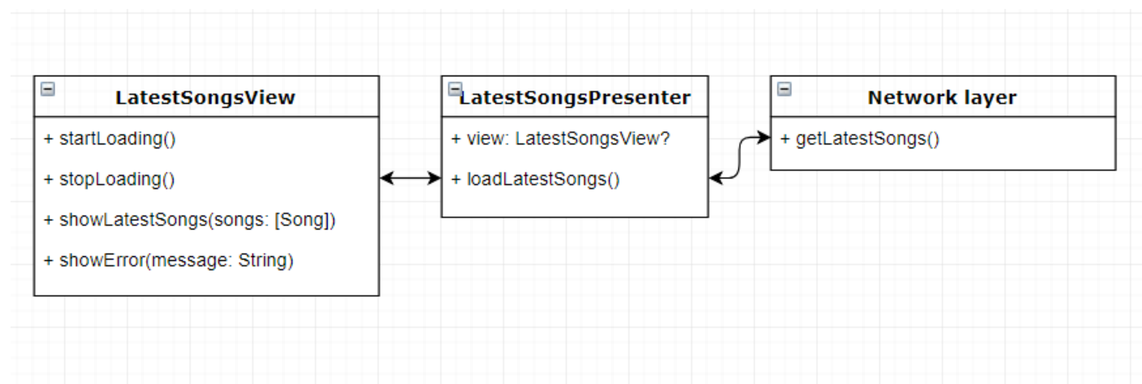

Figure 11. MVP architectural pattern in Musicify application.

The presenter, on the other hand, receives the events emitted by the view and communicates with the model as illustrated in figure 11. Initially, a load-latest-songs event is emitted and sent to the presenter. As a result, the presenter asks the network service to get the latest songs and, at the same time, tells the view to display the loading icon while the networking task is performed. When the task completes, depending on the network response, the presenter returns either the transformed data or an error message to the view and updates the view accordingly.

It is important to note that the `LatestSongsPresenter` class is completely independent of UI controls. The `LatestSongsPresenter` class interacts with `LatestSongsViewController` class through an interface, thus making them loosely coupled with each other. This results in better testability for the presenter. A test case for `LatestSongsPresenter` is described in listing 6.

```
1    class MockLatestSongsView: LatestSongsView {
2        // Implementation of MockLatestSongsView. This will be used to
test the behaviors of Presenter
3    }
4
5    func testLoadLatestSongsSuccessfully() {
6        let expectedSongs = // implementation for expected data
7
8       let actualSongsJSON = // implementation for actual data
9
10       let customEndpointClosure = { (target: MusicifyAPI) -> End-
point in
11           // set up Moya to return stub instead of sending network
requests
12       }
13       // Given
14       let mockMusicifyAPI = MoyaProvider<MusicifyAPI>(endpoint-
Closure: customEndpointClosure, stubClosure: MoyaProvider.immediate-
lyStub)
15       let latestSongsPresenter = LatestSongPresenterImpl(musici-
fyAPIProvider: mockMusicifyAPI)
16       let mockView = MockLatestSongsView()
17       latestSongsPresenter.view = mockView
18       // When
19       latestSongsPresenter.loadLatestSongs()
20       // Then
21
22       // Assert that returned data is correct
23       XCTAssertEqual(mockView.latestSongs, expectedSongs)
24       // Assert that startLoading is called
25       XCTAssert(mockView.didCallStartLoading)
26       // Assert that stopLoading is called
27       XCTAssert(mockView.didCallStopLoading)
```

Listing 6. Test case for LatestSongPresenter class.

Metropolia

As seen in listing 6, `MockLatestSongsView` class adopts and conforms `LatestSongsView` protocol. Therefore, it mimics the behavior of `LatestSongsViewController` class. This helps isolate the behavior of the object under testing, which in this case, is `LatestSong-sPresenter`. Finally, the actual result and expected result are put into comparison to verify the presenter behaves as expected.

3.6 MVVM

First of all, as mentioned in section 2.2.3, the ViewModel is often restricted from knowing anything about the view. In fact, in Musicify application, the `LatestSongsViewModel` does not hold references to the view. Similar to MVP, `LatestSongsViewController` class is also treated as the view in MVVM. It owns the view model and observe any changes through data binding. There are a couple of ways to do so in iOS such as using built-in methods like Key-Value-Observing (KVO), Notifications or using some functional reactive programming libraries like RxSwift[21] or ReactiveCocoa[22]. In Musicify application, simple closures are used as a binding method between the view and the view model.

```
1    class LatestSongsViewModel: LatestSongsViewModelType {
2        // Outputs
3        var latestSongViewModels: [SongViewModel] = [] {
4            didSet {
5                latestSongsDidUpdate?()
6            }
7        }
8        var latestSongsDidUpdate: (() -> Void)?
9        var isLoading: ((Bool) -> Void)?
10       var error: ((_ message: String) -> Void)?
11
12       // Inputs
13       func loadLatestSongs() {
14           isLoading?(true)
15              musicifyAPIProvider.request(.latestSongs) { [weak self]
(result) in
16               switch result {
17               case .success(let response):
18                   let data = response.data
19               guard let songs = try? JSONDecoder().decode([Song].self,
from: data) else {
20                       self?.isLoading?(false)
21                       self?.error?("JSON parsing error")
22                       return
23                   }
24                   self?.isLoading?(false)
```

[21] https://github.com/ReactiveX/RxSwift
[22] https://github.com/ReactiveCocoa/ReactiveCocoa

```
25                    self?.latestSongViewModels = songs.map { SongView-
Model(song: $0) }
26                case .failure(let error):
27                    self?.isLoading?(false)
28                    self?.error?("Connection error")
29                }
30            }
31        }
```

Listing 7. LatestSongsViewModel implementation.

As observed from listing 7, the view model does not know anything about the view. The view model receives load-latest-song event and asks network service to fetch the data for latest songs. Once the task is completed, the view model transforms data into a format which is more convenient for the view. At the same time, it updates the data state inside it and exposes those changes for other objects to subscribe to through binding to its properties: `latestSongViewModels`, `latestSongsDidUpdate`, `isLoading`, and `error`. The view, on the other hand, observes state changes from the view model and updates itself accordingly as seen in listing 8.

```
1    // Bind outputs
2    latestSongsViewModel.latestSongsDidUpdate = { [weak self] in
3        self?.latestSongsTableView.isHidden = false
4        self?.latestSongsTableView.reloadData()
5    }
6    latestSongsViewModel.isLoading = { [weak self] isLoading in
7        if isLoading {
8            self?.loadingIndicator.startAnimating()
9            self?.latestSongsTableView.isHidden = true
10       } else {
11           self?.loadingIndicator.stopAnimating()
12           self?.latestSongsTableView.isHidden = false
13       }
14   }
15   latestSongsViewModel.error = { [weak self] message in
16       self?.latestSongsTableView.isHidden = true
17
18       let alertViewController = UIAlertController(title: "Error",
message: message, preferredStyle: .alert)
19       let cancelAction = UIAlertAction(title: "OK", style: .cancel,
handler: nil)
20       alertViewController.addAction(cancelAction)
21
22       self?.present(alertViewController, animated: true, completion:
nil)
23   }
```

Listing 8. LatestSongsViewController binding function.

The view is notified whenever a change in data state occurs inside `LatestSongsView-Model`. For this reason, latest data always is presented to the users. Therefore, this can result in better user experience and more responsive application.

```
1   // Given
2   let mockMusicifyAPI = MoyaProvider<MusicifyAPI>(endpointClosure:
customEndpointClosure, stubClosure: MoyaProvider.immediatelyStub)
3   let latestSongsViewModel = LatestSongsViewModel(musicifyAPIPro-
vider: mockMusicifyAPI)
4   // When
5   var isLoadingArray: [Bool] = []
6   var didCallLatestSongsDidUpdate = false
7   latestSongsViewModel.isLoading = { isLoadingArray.append($0) }
8   latestSongsViewModel.error = { message in
9       XCTFail("There should NOT be any errors")
10  }
11  latestSongsViewModel.latestSongsDidUpdate = {
didCallLatestSongsDidUpdate = true }
12
13  latestSongsViewModel.loadLatestSongs()
14  // Then
15  XCTAssert(didCallLatestSongsDidUpdate)
16  XCTAssertEqual(latestSongsViewModel.latestSongViewModels, ex-
pectedSongViewModels)
17  XCTAssertEqual(isLoadingArray, [true, false])
```

Listing 9. Test case for ViewModel

Finally, unit testing for `LatestSongsViewModel` class is carried out successfully. Also, as can be observed from listing 9, the mock view is no longer needed as changes in data state can be observed directly from the view model. As a result, the test subscribes to data state in the view model and asserts that output data is expected.

3.7 VIPER

As mentioned in section 2.2.4, there are five components that make up a VIPER module: View, Interactor, Presenter, Entity, and Router. In Musicify application, they are `LatestSongsViewController` class, `LatestSongsInteractor` class, `LatestSongsPresenter` class, `Song` struct and `LatestSongsRouter`, respectively. First of all, the model component is no longer presented in VIPER architectural pattern. Instead, the logic of model has been shifted into Interactor and Entity component. The Entity component is `Song` struct as implemented in section 3.3. On the other hand, `LatestSongsInteractor` class becomes data access layer which manipulates song data and use cases.

Metropolia

```
1   protocol LatestSongsInteractorProtocol {
2       func getLatestSongs(success: (([Song]) -> Void)?, error:
((String) -> Void)?)
3   }
4
5   class LatestSongsInteractor: LatestSongsInteractorProtocol {
6       // MARK: - Dependencies
7       fileprivate let musicifyAPIProvider: MoyaProvider<MusicifyAPI>
8
9       init(musicifyAPIProvider: MoyaProvider<MusicifyAPI> = MoyaPro-
vider<MusicifyAPI>()) {
10          self.musicifyAPIProvider = musicifyAPIProvider
11      }
12
13      func getLatestSongs(success: ((_ songs: [Song]) -> Void)?, er-
ror: ((_ errorMessage: String) -> Void)?) {
14          musicifyAPIProvider.request(.latestSongs) { (result) in
15              switch result {
16              case .success(let response):
17                  let data = response.data
18                  guard let songs = try? JSONDecoder().de-
code([Song].self, from: data) else {
19                      error?("JSON parsing error")
20                      return
21                  }
22                  success?(songs)
23              case .failure:
24                  error?("Connection error")
25              }
26          }
27      }
28  }
```

Listing 10. LatestSongsInteractor implementation

As observed from listing 10, `LatestSongsInteractor` class holds a reference to the network service `MoyaProvider<MusicifyAPI>`. It has one function `getLatestSongs` to ask the network service for song data retrieval and then handles data mapping from JSON to native objects.

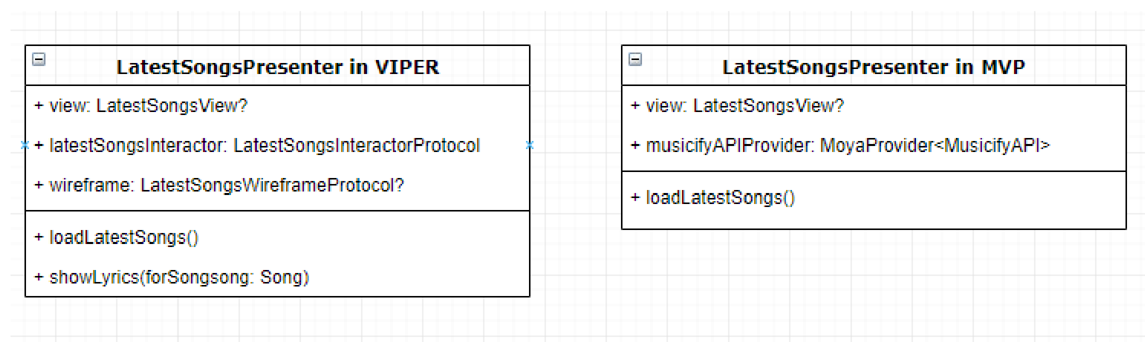

Figure 12. LatestSongsPresenter implementation in VIPER and MVP

Metropolia

The LatestSongsPresenter class in VIPER share some similarities in implementation with the LatestSongsPresenter class in MVP. Firstly, the LatestSongsPresenter in VIPER is also UIKit independent, thus it does not own any UI elements such as UILabel, UIText-Field. Secondly, it interacts with LatestSongsViewController through a protocol called LatestSongsView. This can help facilitate testing as a mock view can be used to verify presenter's behaviors. However, when compared to MVP, the LatestSongsPresenter in VIPER holds a reference to LatestSongsInteractor instead of network service MoyaPro-vider as seen in figure 12. Finally, the presenter has a method called showLyrics which will call the wireframe's method to navigate to a web browser when a song is selected by the users.

```
1   class LatestSongsWireframe: LatestSongsWireframeProtocol {
2       func showLyrics(forSong song: Song) {
3           let lyricsURL = song.lyricsURL
4           UIApplication.shared.open(lyricsURL, options: [:], comple-
tionHandler: nil)
5       }
6
7       class func createLatestSongsModule() -> UIViewController {
8           let interactor = LatestSongsInteractor()
9           let presenter = LatestSongsPresenterImpl(latestSongsIn-
teractor: interactor)
10          presenter.wireframe = LatestSongsWireframe()
11          let viewController = UIStoryboard.init(name: "Main", bun-
dle: nil).instantiateViewController(withIdentifier: "LatestSongsView-
Controller") as! LatestSongsViewController
12          viewController.latestSongsPresenter = presenter
13          return viewController
14      }
15  }
```

Listing 11. LatestSongsWireframe's implementation

As observed from listing 11, the LatestSongsWireframe class has two main functions showLyrics and createLatestSongsModule. Firstly, it handles navigation between the main screen and the web browser in which the song's lyrics are presented. Secondly, the wireframe is responsible for creating the module for the main screen. Therefore, the wireframe sets up its components and wires up their dependencies and delegations.

```
1   func testLoadSongSuccessfully() {
2       // Given
3       let mockMusicifyAPI = MoyaProvider<MusicifyAPI>(endpoint-
Closure: customEndpointClosure, stubClosure: MoyaProvider.immediate-
lyStub)
4       let latestSongsInteractor = LatestSongsInteractor(musici-
fyAPIProvider: mockMusicifyAPI)
5
```

```
6         // When
7         var actualSongs: [Song] = []
8         latestSongsInteractor.getLatestSongs(success: { (songs) in
9             actualSongs = songs
10        }) { (error) in
11            XCTFail("There should not be any errors")
12        }
13        // Then
14        XCTAssertEqual(actualSongs, expectedSongs)
15    }
```

Listing 12. LatestSongsInteractor class's test case

As illustrated in listing 12, similar to MVP, unit testing in a VIPER project is easy to conduct as the responsibilities are more clearly defined and divided into separate classes. With the help of Moya library, a mock network service, which will be used later by the interactor, is configured to return sample data for a list of songs. Next, the method `getLatestSongs` of the interactor gets called and the sample data is returned as a result. Finally, the test passes if the method results in the same outcome as expected.

4 Pattern analysis

In this study, four common architectural patterns that can be used to develop iOS applications have been successfully implemented in the Musicify application to solve the problem of synchronizing the UI with domain data. Table 1 compares these architectural patterns in terms of their distribution of responsibilities, testability, and ease of use. Considerable attention has also been paid to the performance including CPU usage and RAM usage of each pattern.

	MVC	MVP	MVVM	VIPER
Distribution of responsibilities	Entangled	Better	Better	Best
Testability	Hard to test	Better	Better	Best
Ease of use	Easy	Easy	Hard if bindings are carried out with reactive extensions	Hard

Table 1. Pattern-analysis summary. Summarized from Orlov (2015) [11].

In table 1, the distribution of responsibilities describes how responsibilities are divided into different components whereas the testability evaluates if it is easy to write unit test cases. Last but not least, the ease of use evaluates how easy it is for a developer to introduce a new pattern to an iOS project

4.1 Distribution of responsibilities

As observed from table 1, the VIPER architectural pattern seems to have the most balanced distribution of responsibilities amongst the others. Data interaction logic, data presentation logic, data structure logic, and navigation logic are put into separate layers and components. In addition, VIPER is the only one out of four experimenting patterns that solve the routing problem in iOS applications by introducing a new component called Router whose main responsibility is to handle navigation between modules.

Metropolia

On the other hand, as seen in Musicify's MVC implementation, the responsibilities amongst components often become entangled. In reality, the View object is tightly coupled with the controller object in most MVC applications [11]. This tight relationship between the view and the controller is described in figure 13.

Figure 13. Apple's MVC in reality. Copied from Orlov (2015) [11].

It is commonly known that the View and the Controller are often combined into one single object in iOS development, which is the `UIViewController` object. The `LatestSongsViewController` class not only handles networking logic but also does data mapping from JSON to native objects. Additionally, presentation logic such as displaying the latest songs in a table view, displaying loading icon and showing errors to users also resides here. The `LatestSongsViewController` class has acquired some duties that do not necessarily belong to it. As the level of app's complexity increases, maintaining such classes becomes very troublesome. However, there are some additional steps to reduce the complexity of such massive view controllers. Some of which include shifting data processing tasks to another class, introducing data source objects instead of having view controller classes conform to `UITableViewDataSource` protocol and dividing the views into smaller `UIView` subclasses.

MVP and MVVM do have a better distribution of responsibilities than MVC as they introduce new components acting as mediators between the view and the model. The view controller class in both cases is treated as the dumb view, thus its only responsibility is to display data information to the users and propagate user interaction events to the presenter or the view model. However, navigation logic is not addressed in MVP and MVVM. Thus, it often raises a question of which component should be responsible for handling navigation logic in these patterns.

4.2 Testability

In the MVC pattern, most of the application's logic resides in the view controller which is tightly coupled to the UI environment as the `ViewController` class handles the view lifecycles. This results in low testability in MVC pattern. In Musicify's MVC implementation, only unit testing for the `Song` model is carried out with 100% test coverage as observed from table 2.

	MVC	MVP	MVVM	VIPER
Test coverage	Song: 100% ViewController: 78.6%	Song: 100% ViewController: 73.5% Presenter: 100%	Song: 100% ViewController: 83.1% ViewModel: 100%	Song: 100% ViewController: 73.5% Presenter: 100% Interactor: 100%

Table 2. Test coverage of each architectural pattern.

Undoubtedly, VIPER has the best testability out of four patterns thanks to the decoupled nature of VIPER components. Both the presenter and the interactor are UIKit independent, meaning that it is easier to test the business logic inside these two layers. For this reason, the functions inside the `LatestSongsPresenter` class and `LatestSongsInteractor` class are tested with 100% test coverage as seen in table 2. It is worth mentioning that unit testing for `Song` entity is also carried out, thus ensuring that data mapping from JSON to native objects works as expected. However, the number of test cases increases considerably to 8 cases compared to 5 in MVVM and 5 in MVP as there are more layers in a VIPER module. As a result, this may require some extra work to write and maintain unit test cases.

Having better separation of concerns, writing unit tests for the `LatestSongsPresenter` class in MVP and the `LatestSongsViewModel` class in MVVM become more approachable

Metropolia

with 100% test coverage of each class as observed from table 2. It is notable that unit test cases for the view model are often slightly shorter than for the presenter for the same testing behavior as a mock view is no longer needed for the view model in unit testing.

4.3 Ease of use and performance

Despite having high testability and better separation of concerns, VIPER pattern has its own limitations: most notably is the cost of maintainability. As observed from section 3.7, for the same functionality of showing latest songs to users, the VIPER module requires five components namely `LatestSongsViewController` class, `LatestSongsPresenter` class, `LatestSongsInteractor` class, `LatestSongsRouter` class, and `Song` struct compared to only three components in other experimenting patterns. For this reason, having the VIPER pattern implemented in an application can lead to a large number of files. As a result, this requires some extra work to configure many interfaces and classes needed for VIPER module as well as write unit tests for them. Furthermore, one consideration should be taken into account when introducing VIPER to an iOS application as it often requires time and effort for a new developer in a team to learn the basics of VIPER and then familiarize with it.

On the other hand, MVC is the recommended architectural pattern by Apple. For this reason, it can be observed that many lessons on iOS development from Apple are written with MVC pattern in mind [32]. Therefore, developing an iOS application with MVC pattern become easier as developers in a team are familiar with it, even the newcomers.

As MVP and MVVM both have fewer layers than VIPER, learning their basic concepts and implementing it in an application can take less amount of time than VIPER. However, it is worth mentioning that it can be challenging for a new developer to join an MVVM project if there are reactive programming libraries such as RxSwift and ReactiveCocoa involved as some knowledge of reactive functional programming is needed. For this reason, MVP has slightly better ease of use than MVVM.

Performance analysis is done on an iPhone 8 device with the help of Instruments tools. Table 3 shows the CPU usage and RAM usage of each pattern.

	MVC	MVP	MVVM	VIPER
Average CPU Usage	10%	11%	13%	13%
Average RAM Usage	24.2 MB	24.5 MB	24.4 MB	24.4 MB

Table 3. CPU Usage and RAM Usage

As can be observed from table 3, there is no big difference in average RAM usage and average CPU usage of each pattern when implemented in Musicify application. RAM usage is almost identical whereas CPU usage varies slightly from 10% to 13%. Out of four patterns, MVVM and VIPER have the highest CPU usage at 13%, 3% higher than MVC and 2% higher than MVP.

5 Conclusion

The main objective of this thesis was to study and demonstrate some common architectural patterns used in iOS development, namely MVC, MVP, MVVM, and VIPER. As a result, an iOS application called Musicify has been successfully conducted to demonstrate the implementation and the uses of these patterns. Additionally, they were put into comparison in terms of testability, separation of concerns, and ease of development. At the same time, their performances were also acknowledged.

However, as stated by Len Bass, Paul Clements and Rick Kazman (2013) in their book Software Architecture in Practice, there is no good or bad architecture. [3, p.19] In fact, each architectural pattern has its own advantages and drawbacks. Thus, there is no ideal architecture for all iOS applications. For example, VIPER could be a good choice if testability is the architectural primary concern, but it might slow down the development process in the beginning. On the other hand, simple MVC might fit for a project where fast development is the top priority. However, the responsibilities amongst components will become entangled as the level of complexity increases. MVP and MVVM seem to be a potential candidate with better testability and separation of concerns. However, while newcomers may find MVP easier to approach, it would be a challenge for them to apply MVVM if reactive programming libraries such as RxSwift are in use for data binding. Additionally, it is worth mentioning that the navigation problem is not addressed in both MVP and MVVM.

In conclusion, there are various ways to architect an application. An architectural pattern might be suitable for one specific project but not for others. Moreover, it is important to note that apart from distribution of responsibilities, testability, and ease of development, there are many other aspects to be considered when choosing an architectural pattern such as developer skill set, project budget, project deadlines and so on [4, p.46]. Thus, as a mobile developer, it is valuable to know and experience different architectural patterns in order to choose the appropriate one for an application to meet the business needs and goals.

References

1 App Annie. 2017 Retrospective: A Monumental Year for the App Economy. 2017. URL: https://www.appannie.com/en/insights/market-data/app-annie-2017-retrospective. Accessed 25 December 2018.

2 Statista. Number of apps available in leading app stores as of 3rd quarter 2018 [online]. 2018. URL: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Accessed 25 December 2018.

3 Bass L., Clements P. and Kazman R. Software Architecture in Practice. Pearson Education. 2013.

4 Richards M. Software Architecture Patterns: Understand Common Architecture Patterns and When to Use Them. O'Reilly Media. 2015.

5 Open Universiteit. Architectural patterns [online]. 2014. URL: https://www.ou.nl/documents/40554/349790/IM0203_03.pdf. Accessed 5 October 2018.

6 Tutorialspoint. MVC Framework - Introduction [online]. 2019. URL: https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm. Accessed 30 January 2019.

7 Apple Inc. Concepts in Objective-C Programming [online]. 2012. URL: https://developer.apple.com/library/content/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html. Accessed 28 November 2017.

8 Potel M. MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java. Taligent, Inc. 1996.

9 Almiray A. MVC Patterns [online]. 2015. URL: http://aalmiray.github.io/griffon-patterns. Accessed 31 January 2019.

10 ThinkMobiles. iOS architecture patterns: A guide for developers [online]. URL: https://thinkmobiles.com/blog/ios-architecture-patterns/. Accessed 31 January 2019.

11 Orlov B. iOS Architecture Patterns [online]. 2015. URL: https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52. Accessed 31 January 2019.

12 Microsoft. The MVVM Pattern [online]. 2017. URL: https://msdn.microsoft.com/en-us/library/hh848246.aspx. Accessed 31 January 2019.

13 Cacheaux R and Berlin J. Advanced iOS App Architecture – Real-world app architecture in Swift 4.2. Ray Wenderlich. 2018.

14 Martin R. The Clean Architecture [online]. 2012. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Accessed 31 January 2019.

15 Gilbert J. and Stoll C. Architecting iOS Apps with VIPER [online]. 2014. URL: https://www.objc.io/issues/13-architecture/viper/. Accessed 31 January 2019.

16 Apple Inc. XCode [online]. 2019. URL: https://developer.apple.com/xcode/. Accessed 31 January 2019.

17 Apple Inc. Interface Builder Built-In [online]. 2019. URL: https://developer.apple.com/xcode/interface-builder/. Accessed 31 January 2019.

18 Apple Inc. Understading Auto Layout [online]. 2019. URL: https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/index.html. Accessed 31 January 2019.

19 Apple Inc. XCTest [online]. 2019. URL: https://developer.apple.com/documentation/xctest. Accessed 31 January 2019.

20 DZone. Pros and Cons of Using XCTest for iOS Testing [online]. 2016. URL: https://dzone.com/articles/pros-and-cons-of-using-xctest-for-ios-testing. Accessed 31 January 2019.

21 Apple Inc. About Simulator [online]. 2018. URL: https://developer.apple.com/library/archive/documentation/IDEs/Conceptual/iOS_Simulator_Guide/Introduction/Introduction.html. Accessed 31 January 2019.

22 Apple Inc. Testing and Debugging in Simulator [online]. 2018. URL: https://developer.apple.com/library/archive/documentation/IDEs/Conceptual/iOS_Simulator_Guide/TestingontheiOSSimulator/TestingontheiOSSimulator.html#//apple_ref/doc/uid/TP40012848-CH4-SW1. Accessed 31 January 2019.

23 Apple Inc. Instruments Overview [online]. 2018. URL: https://help.apple.com/instruments/mac/current/#/dev7b09c84f5. Accessed 31 January 2019.

24 Apple Inc. Swift [online]. 2019. URL: https://developer.apple.com/swift/. Accessed 31 January 2019.

25 Apple Inc. About Swift [online]. 2018. URL: https://docs.swift.org/swift-book/. Accessed 31 January 2019.

26 Stackoverflow. Developer Survey 2015 [online]. 2015. URL: https://insights.stackoverflow.com/survey/2015#tech-super. Accessed 31 January 2019.

27 Apple Inc. Codable [online]. URL: https://developer.apple.com/documentation/swift/codable. Accessed 31 January 2019.

28 Cocoapods. Cocoapods [online]. 2015. URL: https://cocoapods.org/. Accessed 31 January 2019.

29 Apple Inc. About Networking [online]. 2004. URL: https://developer.apple.com/library/archive/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/Introduction/Introduction.html#//apple_ref/doc/uid/TP40010220-CH12-BBCFIHFH. Accessed 31 January 2019.

30 Moya. Moya [online]. URL: https://github.com/Moya/Moya. Accessed 31 January 2019.

31 Flower M. Mocks Aren't Stubs [online]. URL: https://martinfowler.com/articles/mocksArentStubs.html. Accessed 31 January 2019.

32 Apple Inc. Start Developing iOS Apps (Swift) [online]. URL: https://developer.apple.com/library/archive/referencelibrary/GettingStarted/DevelopiOSAppsSwift/. Accessed 31 January 2019.

Musicify's Github repository

Github: https://github.com/akzuki/Musicify

MVC version: https://github.com/akzuki/Musicify/tree/MVC

MVP version: https://github.com/akzuki/Musicify/tree/MVP

MVVM version: https://github.com/akzuki/Musicify/tree/MVVM

VIPER version: https://github.com/akzuki/Musicify/tree/VIPER

Full implementation of MusicifyAPI network layer using Moya

```
1   //
2   //  MusicifyAPI.swift
3   //  Musicify
4   //
5   //  Created by Hai Phan on 02/01/2019.
6   //  Copyright © 2019 Hai Phan. All rights reserved.
7   //
8
9   import Foundation
10  import Moya
11
12  enum MusicifyAPI {
13      case latestSongs
14  }
15
16  extension MusicifyAPI: TargetType {
17      var baseURL: URL {
18          return URL(string: "https://api.myjson.com")!
19      }
20
21      var path: String {
22          switch self {
23          case .latestSongs:
24              return "/bins/183jfg"
25          }
26      }
27
28      var method: Moya.Method {
29          switch self {
30          case .latestSongs:
31              return .get
32          }
33      }
34
35      var task: Task {
36          switch self {
37          case .latestSongs:
38              return .requestPlain
39          }
40      }
41
42      var headers: [String : String]? {
43          return nil
44      }
45
46      var sampleData: Data {
47          switch self {
48          case .latestSongs:
49              return (
50                  """
51                  [
52                      {
53                          "id": 1,
54                          "name": "Back To You",
```

```
55                          "artist": "Selena Gomez",
56                          "releaseYear": "2018",
57                          "thumbnailURL":
"https://i.ytimg.com/vi/VY1eFxgRR-k/maxresdefault.jpg",
58                          "lyricsURL":
"https://www.azlyrics.com/lyrics/selenagomez/backtoyou.html"
59                      },
60                      {
61                          "id": 2,
62                          "name": "Dance To This",
63                          "artist": "Troye Sivan ft Ariana Grande",
64                          "releaseYear": "2018",
65                          "thumbnailURL":
"https://i.ytimg.com/vi/bhxhNIQBKJI/maxresdefault.jpg",
66                          "lyricsURL":
"https://www.azlyrics.com/lyrics/troyesivan/dancetothis.html"
67                      },
68                      {
69                          "id": 3,
70                          "name": "Lost In Japan",
71                          "artist": "Shawn Mendes",
72                          "releaseYear": "2018",
73                          "thumbnailURL":
"https://i.ytimg.com/vi/ycy30LIbq4w/maxresdefault.jpg",
74                          "lyricsURL":
"https://www.azlyrics.com/lyrics/shawnmendes/lostinjapan.html"
75                      },
76                      {
77                          "id": 4,
78                          "name": "My My My!",
79                          "artist": "Troye Sivan",
80                          "releaseYear": "2018",
81                          "thumbnailURL":
"https://i.ytimg.com/vi/k5TqNsr6YuQ/maxresdefault.jpg",
82                          "lyricsURL":
"https://www.azlyrics.com/lyrics/troyesivan/mymymy.html"
83                      }
84                  ]
85                  """
86                  .data(using: .utf8)!
87              )
88          }
89      }
90
```

Project file structure

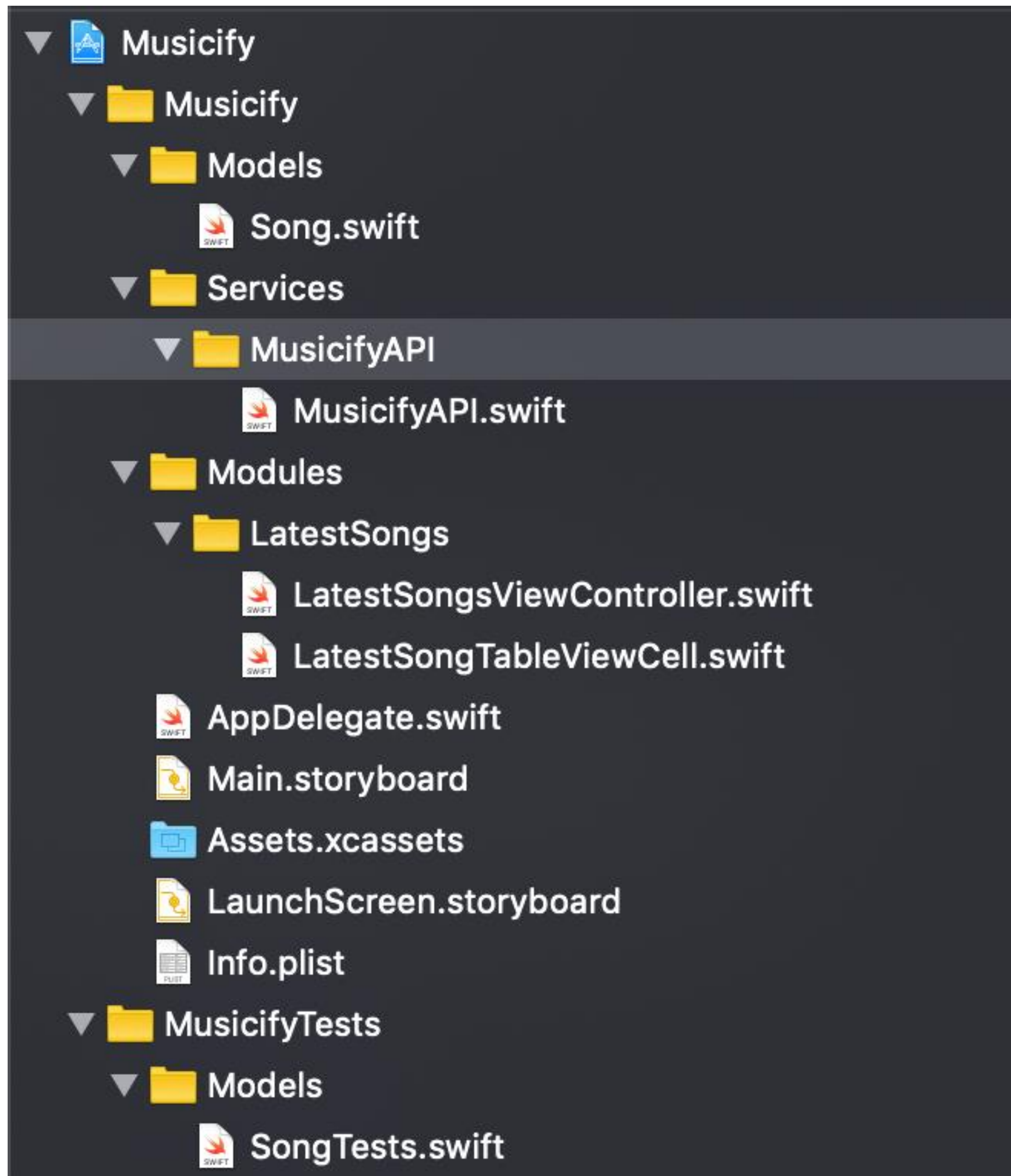

Figure 14. Project file structure for MVC

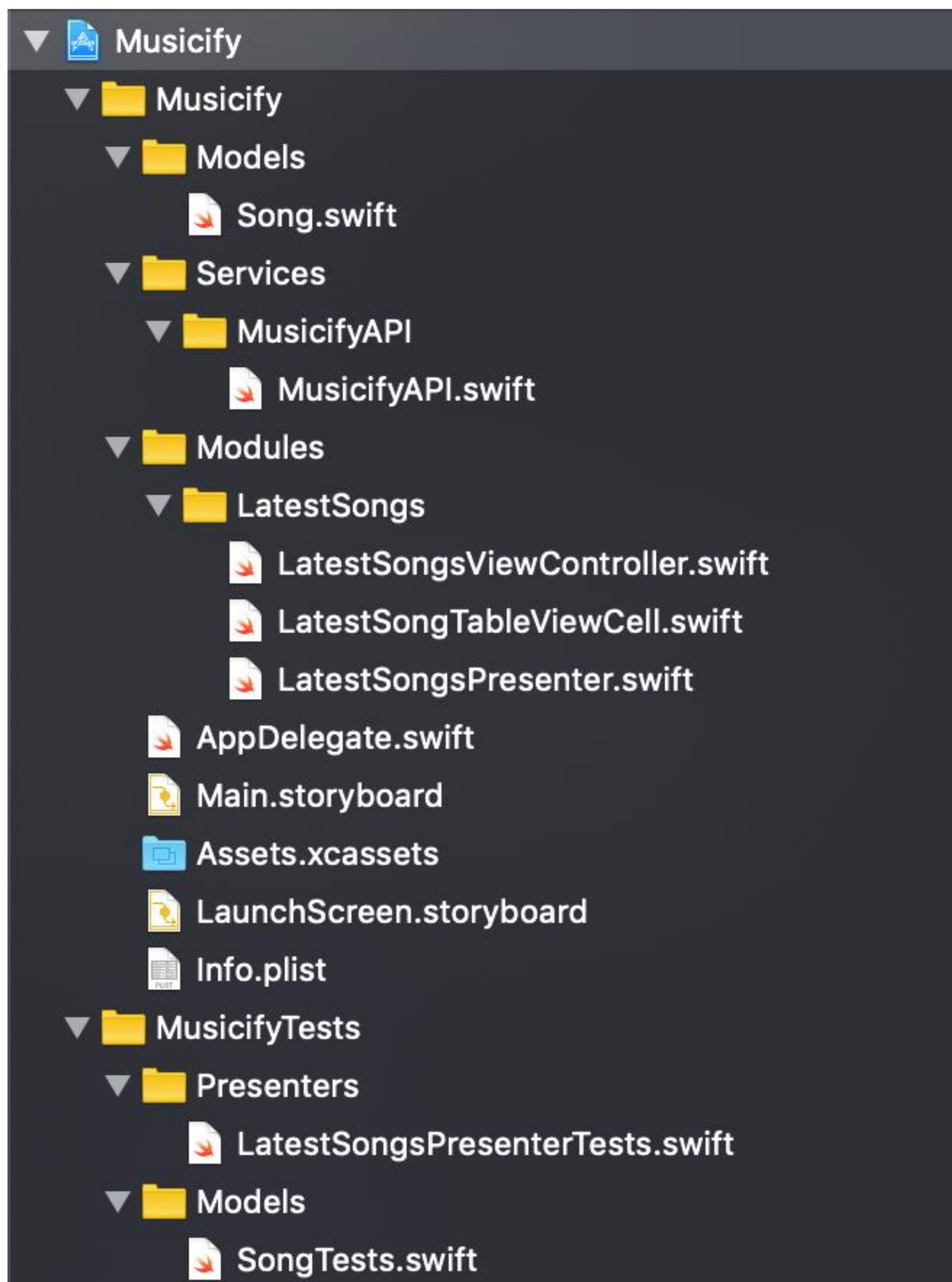

Figure 15. Project file structure for MVP

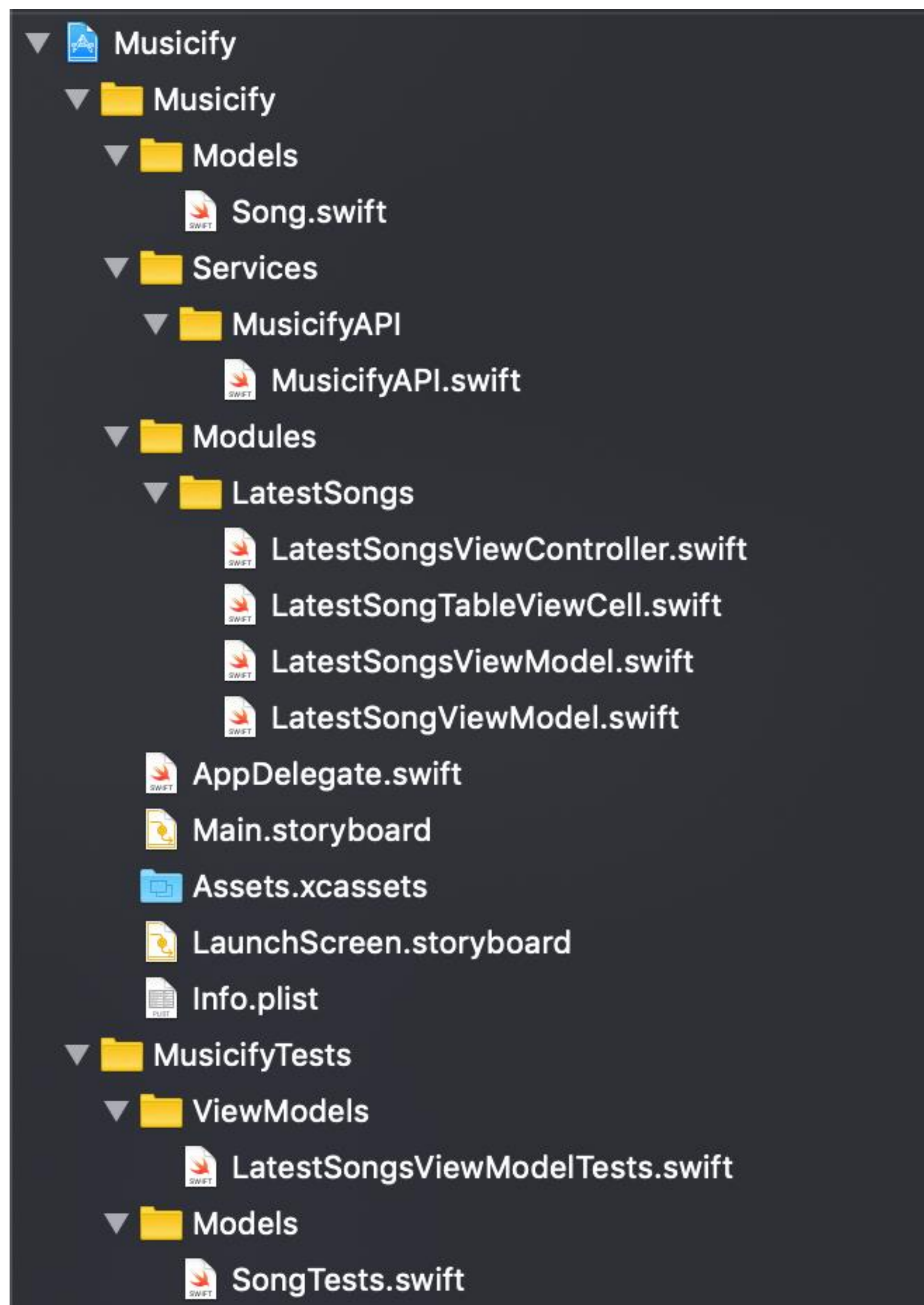

Figure 16. Project file structure for MVVM

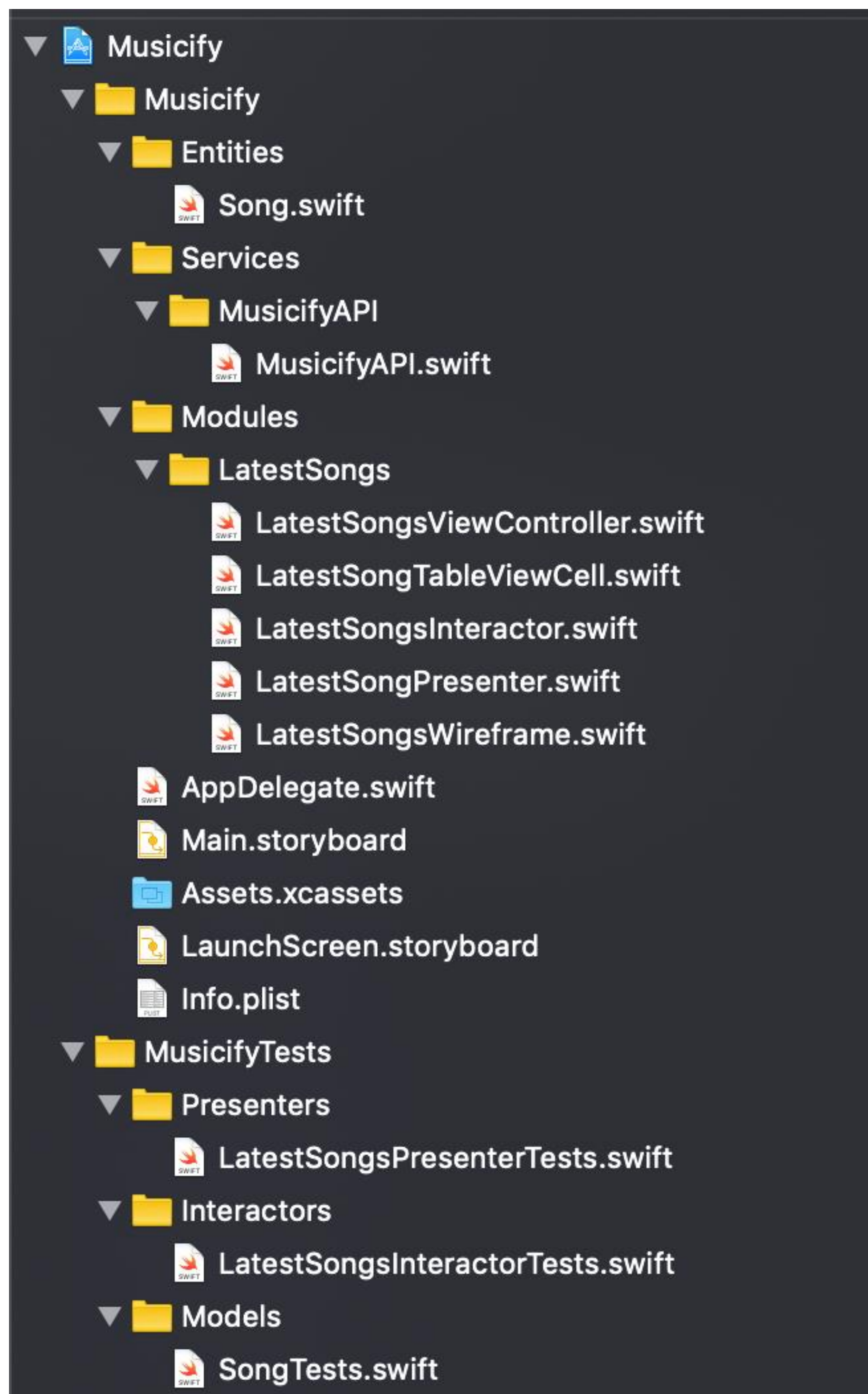

Figure 17. Project file structure for VIPER