Tung Duc Vo

Application of protocol-oriented programming in iOS development

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Thesis

6 April 2019

Author(s) Title Number of Pages Date	Tung Duc Vo Application of protocol-oriented programming in iOS development 38 pages 6 April 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Mobile Software Development
Instructor(s)	Peter Hjort, Senior Lecturer

There has always been a demand for better approach to build scalable and extensible software for tech companies and communities. For Cocoa developers, architectural design pattern is one of the most popular topics in tech blogs, talks and events. Since the introduction of Swift and protocol-oriented programming paradigm, Cocoa developers have another option to build better software.

This study aims to clarify the concepts of protocol-oriented programming and its advantages over its counterpart, object-oriented programming, for building software using Swift language. Moreover, this study also presents different techniques and practices for improving the scalability and maintainability of the software. The thesis also introduces how to use Model-View-View Model architectural design pattern as a replacement of the traditional Model-View-Controller.

The project used to demonstrate the mentioned techniques is written in Swift and developed for iOS devices. Created solely for the purpose of this thesis, the application is not published in Apple App Store. However, the project is open-source and can be found in Github.

Overall, the main purpose of the thesis is to recommend a collection of techniques, together with protocol-oriented design that can be used to create better software.

Keywords	POP, unit testing, ios, mobile development, MVVM, Swift

Contents

Abbreviations

MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-View Model
VIPER	View-Interactor-Presenter-Entity-Router
POP	Protocol-oriented programming
OOP	Object-oriented programming
TDD	Test-driven development
WWDC	Apple Worldwide Developers Conference
Rx	Reactive Extensions

Glossary

iOS	An operating system used for Apple mobile devices (iPhones and iPads).
Cocoa	Apple's native object-oriented application programming interface.
Github	A web-based hosting service for version control using Git.
OSX	An operating system used for mobile devices manufactured by Apple Inc.
Separation of concern	A design principle for separating a computer program into distinct sections, and each section addresses a separate concern.
Unit testing	A level of software testing where individual units/ components of a software are tested

1 Introduction

Technology has been evolving so fast in the last decades that it is not easy to keep up with it. The way we communicate, the way we learn, the way we entertain, and many of our daily routines have been affected by tech products. As a result, the demand for building more scalable and extensible software for tech products has been growing among companies and tech communities. Similarly, for Cocoa programmers, different approaches for building better iOS or OSX applications have been a never-ending discussion in programming websites, tech blogs, etc. Therefore, apart from traditional Model-View-Controller (MVC), many new architectural design patterns have been introduced to Cocoa programmers' community. Some of them can be listed such as Model-View-View Model (MVVM) and View-Presenter-Interactor-Router-Entity (VIPER) [1].

However, only until the introduction of Swift programming language, new programming paradigm is available for Cocoa programmers. Swift programming language was introduced by Apple in 2014, and a year later they announced Swift 2 and that Swift is the first protocol-oriented programming (POP) language ever created [2].

Since then, Apple have been promoting POP as the new way of programming and writing applications when working with Swift. For Cocoa software development using Swift language, POP is a better option in terms of performance, readability, testability and scalability. This study aims to demonstrate how POP is applied to develop iOS application with better performance and software design.

Besides POP, this study also introduces MVVM architectural design pattern. For iOS and OSX development, MVC is the most well-known architectural design pattern among developers as it is frequently recommended and promoted by Apple [3]. This pattern, however, has several disadvantages as the software scales. One of the main drawbacks of MVC is separation of concerns as view controllers often have too many responsibilities. MVVM, however, by introducing extra components, distributes part of the responsibilities among them. The key component, View Model, as its name suggests, is a presentation model of the View. View model encapsulates the representational data creation complexity within itself, then notify the View update via data binding, thus reduces responsibilities from the View controller.

2 Theoretical Background

2.1 Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm and its practices can help to create large software with reusable and scalable components [4]. OOP is one of the most essential steps for most software developers in the past decades or more. As its name suggested, the main principle behind OOP is to create software based on objects. Objects are instances of class and in an OOP program, objects are designed interact with each other.

Nowadays, many of the most commonly used programming languages such as Java, C++, C#, Python, PHP, JavaScript, Objective-C, Swift support OOP [5].

2.1.1 Class

Class is the fundamental component of OOP. It is a blueprint of creating objects, providing member data and member methods [6]. An object is a specific instance created from a particular class.

```
class Student {

    var studentNumber: String
    var socialID: String
    var fullName: String
    var birthdate: Date
    var email: String
    var address: String

    init(studentNumber: String,
         socialID: String,
         fullName: String,
         birthdate: Date,
         email: String,
         address: String) {
        self.studentNumber = studentNumber
        self.socialID = socialID
        self.fullName = fullName
        self.birthdate = birthdate
        self.email = email
        self.address = address
    }

    func orderTranscript() {
        // Implementation
    }

    func enrollToCourse(_ course: Course) {
        // Implementation
    }
}
```

Figure 1. Example of Student class

As shown in Figure 1, class `Student` has six member variables and two member methods.

2.1.2 Reference type and value types

In Swift and most OOP languages, data types can be divided into two categories: reference type and value type. There are many differences between the two types, but the primary difference is how their instances are passed [4].

2.1.2.1 Reference types

Reference type contains pointer to memory address where the actual data is held. Instances of reference types share a single copy of data [7]. Passing an instance of a reference type is passing a reference of the original instance. As the result, both references point to the same instance; therefore, updating any instace data reflects the changes in the others. Swift reference types include *class, functions* and *closures*.

2.1.2.2 Value types

An instance of a value type holds its own data in a separate memory location [7]. When an instance of value type is passed, it is actually passing a copy of its own data. Therefore, updating an instance does not reflect the changes to any others. There are many kinds of value types in Swift such as *struct*, *enum*, *tuples* and other primitive types (*Int, Double, Array, Dictionary, etc.*) [8].

2.1.3 Objective-C vs Swift

Swift language was introduced by Apple in 2014. Since then, Swift has been slowly taking over Objective-C as the preferred programming language for Cocoa development. This modern language is considered easier to read and learn than its counterpart, and also requires less code which in turn improves the development process and allows developers more to be productive [9].

Swift has been built as a protocol-oriented and object-oriented language. For that reason, there are several differences in the primitive data types between Objective-C and Swift which are essential for the development between the two languages. Primitive data types such as integer, double, float, bool are value types in both languages. Other built-in types such as string, array, dictionary, while being reference types in Objective-C, are implemented as struct in Swift and are value types [8]. The following tables show the similarities and differences between the two languages in terms of data types.

Data	**Objective-C**	**Swift**

Integer	Value type	Value type
Double	Value type	Value type
String	Reference type	Value type
Array	Reference type	Value type
Enum	Value type	Value type
Dictionary	Reference type	Value type
Set	Value type	Value type
Struct	Value type	Value type
Tuple	Not available	Value type
Closure/block	Reference type	Reference type
Function	Reference type	Reference type
Class	Reference type	Reference type

Table 1. Data type comparision between Swift and Objective-C

2.1.4 Object-oriented programming features

The concept of OOP associates with class and its features. The most essential features of this programming paradigm include [10]:

- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

2.1.4.1 Encapsulation

Data encapsulation is a fundamental feature of class. The purpose is to prevent direct access to the object's states and data. Instead, class provides certain methods for the outsiders to communicate with, update or modify the data within the object [11]. In Swift, **private** or **fileprivate** can be used to keep its states private within a code scope or a file, and **internal** can be used to avoid access from different module. On the other hand, **public** and **open** are used to give global access to properties and methods.

```
class Car {
    let year: Int
    let model: String
    private var mileage: Double = 0

    init(year: Int, model: String) {
        self.year = year
        self.model = model
    }

    func currentMileage() -> Double {
        return mileage
    }

    func travel(distance: Double) {
        mileage += distance
    }
}
```

Figure 2. Example of encapsulation

In figure, the outside world cannot modify `Car` objects' mileage directly since mileage is defined as a private property. Instead, methods `travel(distance:)` and `current-Mileage()` are provided to update and get current mileage.

2.1.4.2 Inheritance

Inheritance is the process by a class derive properties and characteristics of another class. The concept of inheritance improves the reusability and extensibility of class, by allowing additional functionalities to be added to an existing class. A class which is inherited from is called a super class or base class and the other is called a subclass. Inheritance allows classes to be created and built upon existing classes [12].

```
enum Gender {
    case male
    case female
}

class Person {
    let firstName: String
    let lastName: String
    let gender: Gender
    let birthdate: Date
    let socialID: String
    let addresss: String

    init(firstName: String, lastName: String, gender: Gender, birthdate: Date, socialID: String, addresss: String) {
        self.firstName = firstName
        self.lastName = lastName
        self.gender = gender
        self.birthdate = birthdate
        self.socialID = socialID
        self.addresss = addresss
    }
}

class Student: Person {
    let studentID: String

    init(firstName: String, lastName: String, gender: Gender, birthdate: Date, socialID: String, addresss: String, studentID: String) {
        self.studentID = studentID
        super.init(firstName: firstName, lastName: lastName, gender: gender, birthdate: birthdate, socialID: socialID, addresss: addresss)
    }
}
```

Figure 3. Example of class inheritance

In Figure 3, Student subclasses Person and inherits all of its parent's properties and initializer.

2.1.4.3 Abstraction

The key concept of abstraction is handling complexity of objects by hiding its unnecessary details [13]. In software development, program can grow large and objects can have enormous number of recursive states and data. In addition to that, objects communicate with other objects which makes maintaining and changing the program even more challenging. Abstraction is the process of hiding irrelevant characteristics of the object in order to reduce the complexity and increase the proficiency.

This approach hides all the internal implementation details and only reveal the operation relevant for the other objects. Swift offers abstraction with protocols. Protocols will be discussed in more detail in later parts of this study.

2.2 Protocol-oriented programming

The concept of using protocol and value-oriented programming with Swift was introduced in Apple's Worldwide Developer Conference 2015. As Apple stated in the event,

Swift is the first POP language ever. It has many built-in features that most programming languages do not in order to make protocol-oriented possible. As its name suggests, protocol is the fundamental component in this programming paradigm. It is, however, much more than just beginning the design thinking with protocols rather than a class. Apple also stated that it is preferable to use value types over reference types where appropriate [14].

2.2.1 Object-oriented design problems

Object-oriented design and its feature are vastly adopted by many modern programming languages such as Python, Ruby, Scala, Java, etc. Objective-C, the only official language used in Cocoa development before Swift introduction, is also an OOP language. That said, object-oriented design has been the most popular programming paradigm to Cocoa developers. There are, however, several drawbacks in the design which increases the complexity and vulnerability of the software. The three most well-known ones, as discussed at Apple Worldwide Developers Conference (WWDC) 2015, are [14]:

- Implicit sharing of mutable state
- Business-depended inheritance
- Loss of type relationship

2.2.1.1 Implicit sharing of mutable state

Since instances of reference types share a single copy of data, sharing is implicit. Updating a single instance reflects the changes in all other shared instances, and this could lead to undesirable outcomes in cases where the update is expected to take place on one single instance. Consequently, in OOP, defensive copying is brought into use. Defensive copying is a technique used to avoid unwanted effects caused by modifications of shared objects [15]. In this process, a new object is created, and its data is identical to the original object's. However, this approach, when overused, slows down the software because of object creation and memory pressure. In computing science, creating an object takes place on Heap memory instead of Stack memory. Heap is used for dynamic memory allocation and Stack for static memory allocation. The Stack is a "LIFO" (last in, first out) data structure which is managed and optimized by the CPU efficiently [16]. For that reason, creating and accessing stack variables are fast.

On the other hand, because Heap is used for dynamic memory allocation, heap elements can be accessed at any random time. This increase the complexity to manage which part of Heap memory are allocated or freed at a given time.

In a multi-threaded situation, a new Stack is created for every individual thread, but all threads share the same Heap. Stack is thread specific and Heap is application specific. Reference instances can be accessed different threads, and if not handled properly can lead to unexpected situations such as deadlocks and race conditions. Thus, multi-threading techniques such as locks, mutexes, synchronizations are used to avoid ensure the correctness of the program. In consequence, it increases the overall complexity of the software.

2.2.1.2 Business-depended inheritance

As mentioned above, inheritance, the fundamental feature of OOP, is a mechanism of basing a class upon another super class. In Swift as well as Objective-C, it only allows single inheritance, meaning that any class cannot inherit from more than one super class [17]. Single inheritance introduces several challenges in object-oriented design.

First of all, class inheritance is intrusive. Base class is required to be well-chosen and defined before implementing the derived class, and thus makes it more difficult to update base class in the future. Updating base class while maintaining the correctness of the program is challenging.

Secondly, the derived class has to accept all the properties and functionalities from its base class even if they are not necessary. This also leads to initialization burden as derived class has to instantiate all the required properties from base class.

Moreover, derived class has to guarantee that it does not break the invariants. Derived classes have the possibility to override their base class's implementation. This enables the extensibility but also exposes to the risk that the original logic is altered incorrectly and leads to software bug. Thus, programmers have to write implementation of derived class with caution and in accordance with base class requirements.

2.2.1.3 Lost type relationships

In Swift, Class is not an optimal option where type relationship matters. Superclass does not know the exact type when it is sub-classed. In addition, in Class, methods require implementation. Thus, methods that return value and require final implementation from subclass is forced to have default implementation. In addition, Class cannot express crucial type relationship between the type of self and the type of other. Using Class removes important type relationship because of lack of abstraction.

```
class Shape {
    func area() -> CGFloat {
        fatalError("Must be overriden")
    }

    func isEqual(to shape: Shape) -> Bool {
        fatalError("Must be overriden")
    }
}

class Rectangle: Shape {
    var width: CGFloat
    var height: CGFloat

    init(width: CGFloat, height: CGFloat) {
        self.width = width
        self.height = height
    }

    override func area() -> CGFloat {
        return width * height
    }

    override func isEqual(to shape: Shape) -> Bool {
        if let shape = shape as? Rectangle {
            return width == shape.width && height == shape.height
        }

        return false
    }
}
```

Figure 4. Example of Shape implementation using class.

```
protocol Shape: Equatable {
    func area() -> CGFloat
}

struct Rectangle: Shape {
    var width: CGFloat
    var height: CGFloat

    init(width: CGFloat, height: CGFloat) {
        self.width = width
        self.height = height
    }

    func area() -> CGFloat {
        return width * height
    }
}
```

Figure 5. Example of Shape implementation using protocol and struct

Figure 4 and Figure 5 show two different approaches of implementing `Shape`. The first approach which uses class and class inherent shows several flaws in the design. First of all, the base class `Shape` has to provide default error implementation in order to implicitly enforce sub classes to provide actual implementation. Sub-classing `Shape` without overriding the two methods will crash the software. Secondly, in class `Rectangle`, the overridden method `isEqual(to: )` accepts any sub types of `Shape`, which should not be the case. This is a sign of lost type relationship.

The second approach, as seen in Figure 5, uses protocol and struct to solve the problem. Firstly, protocol does not provide any implementation of the method and protocol forces any type conforming it to do the job. Secondly, by using Self in the method declaration, it enforces the parameter type to be the same as the one calling the method. Thus, it gives a stronger type relationship and the intention of the code. Last but not least, using value semantics for `Rectangle` is more appropriate in this case as different rectangles, even not sharing same memory address, should be treated identically when they share the same width and height.

2.2.2 Solution with protocol-oriented programming

Thanks to Swift's characteristics, the problems discussed in section 2.2.1 are minimalized with protocol-oriented design.

2.2.2.1 Protocols

A protocol is a set of requirements for methods, properties, initializers that represent a specific task or functionality. Any type that conforms the protocol is forced to provide an actual implementation of those requirements [18]. The conforming type can be either a class, struct or enumeration.

The basic concept of protocol is similar to interface in Java but in Swift, protocol has additional features that enable Swift to be a POP language. Due to the usage of protocol and value types, limitations from OOP and classes are handled effectively. The benefits of protocol include [18]:

- Support both value types and reference types
- Support static and dynamic type relationship
- Support retroactive modeling
- Not impose instance data on models
- Not impose initialization on models
- Make clear what to implement and reduce ambiguity

2.2.3 Features of protocols

Basic feature of protocol is specifying a set of requirements such as properties, instance or static methods and initializers. In Swift, protocols are much more than that. It has many additional features which make protocols much more powerful. The most significant ones are protocol extension, protocol inheritance and protocol composition.

2.2.3.1 Protocol extension

Protocol extension can be used to provide default implementation of methods or computed properties to all or multiple conforming types [18]. The main benefit is that behavior is applied on the protocol instead of having to provide implementation in each individual type conformance.

```
protocol AudioPlayer {
    var isPlaying: Bool { get }
    func setUp(with item: AudioPlayerItem)
    func play()
    func pause()
}

extension AudioPlayer {
    func toggleState() {
        if isPlaying {
            pause()
        } else {
            play()
        }
    }
}
```

Figure 6. Example of protocol extension.

From Figure 4, protocol extension is used to provide implementation of method `toggleState()` to protocol `AudioPlayer`. Therefore, any types conforming `AudioPlayer` automatically share default implementation of `toggleState()`.

2.2.3.2 Protocol inheritance

The concept of protocol inheritance is similar to class inheritance. A protocol can inherit one or more protocols and can add additional requirements on top of the requirements it inherits [18]. However, unlike class inheritance in which a class can only inherit one base class, a protocol can inherit multiple protocols at the same time. That enables type to break down into smaller components, and thus improves code reusability. The syntax of protocol inheritance is similar to class inheritance's:

```swift
protocol ReadPermission {
    func readFile(from url: URL, completion: ((Data) -> Void)?)
}

protocol WritePermission {
    func writeFile(data: Data?, to url: URL, completion: ((Data) -> Void)?)
}

// Protocol inheritance
protocol AdminPermission: ReadPermission, WritePermission {
    func grantPermisson(_ permission: FilePermission, to user: User)
}
```

Figure 7. Example of protocol inheritance

From Figure 7, `AdminPermission` inherits from `ReadPermission` and `WritePermission`, making any type conforming `AdminPermission` needs to provide implementation for all three methods readFile(from url: URL, completion: ((Data) -> Void)?), `writeFile(data: Data?, to url: URL, completion: ((Data) -> Void)?)` and

.

2.2.3.3 Protocol composition

Protocol composition is the process of combining one or more protocols to form a new set of requirements for a type without defining any new protocol [18]. Protocol composition can be used to solve limitation with class inheritance. As mentioned earlier, Swift only supports single inheritance, thus adopting traits from two distinct classes is problematic. In this case, composing the traits and making it a brand-new type is better approach. For example, it would be impossible to reuse class `House` and `Car` to create `CamperVan`. However, `CamperVan` can be built by combining `HousingTrait` and `VehicleTrait`.

```
struct CardReader {
    var scanner: CustomerCardScanner & CreditCardScanner
}
```

Figure 8. Example of protocol composition.

As illustrated in the figure, variable scanner has type as protocol composition from `CustomerCardScanner` and `CreditCardScanner`. Protocol composition is specified by separating multiple protocols with ampersands (&). Besides its list of protocols, one class type can also be added to the composition, which in turn specifies the required base class.

2.2.4 Generics

Generic programming is the way of writing flexible and reusable code that can work with multiple types. Numerous related or unrelated types can share common generic code, that helps to avoid duplications and express the functionalities in a clear, abstracted manner [19].

Generics is a powerful feature of Swift as it enables much of the Swift standard library to be built upon it. For example, in Swift, Array and Dictionary both share generic collections' behaviors. Therefore, as for Array, it can hold a collection of integers, strings

or any other types. In this case, generics is used to create generic types. Similarly, it can also be used to write generic functions which can work with multiple types and treat them identically.

There are multiple ways to achieve generics in Swift, including [19]:

- Generic functions
- Generic types
- Associated types in protocol
- Generics `where` clauses

2.2.5 Design patterns

In software engineering, a design pattern a reusable solution to a problem encountered when it comes to software design [20]. Design pattern offers a general approach to solve a specific set of similar problems. Programmers can adopt different design patterns as best practices in order to solve software or system design problems.

In short, design patterns can be used anywhere in software to obtain an optimal solution to complex software problems. Design patterns are not specifically designed for POP paradigm. There are, however, some design patterns that can significantly benefit a protocol-oriented software. This study will introduce the most commonly used and beneficial ones such as composite, delegation and dependency injection.

2.2.5.1 Composite design pattern

Composite pattern is a design pattern in which a collection of objects is handled the same way as any single object of the collection. In composite design pattern, related objects are composed together into a tree structure that represents part-whole hierarchies. Composite design pattern is used when the difference between compositions of objects and individual objects should be neglected [20].

2.2.5.2 Delegation design pattern

Delegation is a communication design pattern in which one object order another object to react to a certain event or act on behalf of it. In this design pattern, the delegating object often has the reference of the delegate object and sends a message to it at appropriate time[21]. The message contains a specific information and requires the delegate object to handle. Or the delegating object may ask for more information within the current context so that it can continue executing the program. The main benefit of delegation is that it enables a chain of responsibilities in a central context to be distributed among separated entities.

Delegation is one of the most used design patterns in Cocoa frameworks. It can be seen anywhere from common classes in UIKit framework such as `UITableView`, `UICollectionView`, `UITextField`, `UINavigationController` to classes in other frameworks like `NSFetchedhResultsController`, `NSURLSession`, etc. Typically, delegating object only keeps a week reference to the delegate object to avoid retain cycle.

Frequently, when working with delegation design pattern, protocols are used to define the set of required methods to delegate objects. The following figure is an example of how delegation is adapted in `UITableView` from UIKit framework:

```
class UITableView: UIScrollView {

    weak open var dataSource: UITableViewDataSource?

    weak open var delegate: UITableViewDelegate?

}

protocol UITableViewDataSource : NSObjectProtocol {
    // Required methods are defined
}

protocol UITableViewDelegate : UIScrollViewDelegate {
    // Required methods are defined
}
```

Figure 9. Example of delegation design pattern

2.2.5.3 Dependency injection design pattern

The core idea of dependency injection is to have one object provides dependency to another object. The dependent object uses the supplied dependency as a service and the service is made as part of the object's state [22]. The passing of the dependency to the dependent object is called an injection. The essential requirement of this technique is to pass the service rather than allowing the client to create its own dependency.

There are many different approaches to introduce a dependency to a client. The three most common methods are setter-, interface- and constructor-based injection. The main difference between setter and constructor injection is when the dependency is passed can be used. On the other hand, interface injection gives the dependency a chance to manipulate the injection. In this method, a dependency provides its own method implementation to introduce itself to a client [23].

1. constructor injection: an initializer is used to inject the dependency.

2. setter injection: a setter method is used to inject the dependency.

3. interface injection: an injector method provided by the dependency is used to introduce the dependency to a client. The client is required to expose a setter method that receives the service.

Dependency injection is one of the most common method used in protocol-oriented design. The dependency is commonly defined as a protocol, meaning that its concrete type is dynamic and completely dependent on the injector. Applying this approach, replacing a dependency with a new object happens only at the injector's implementation, and thus prevents modification at the client. In addition, another benefit of dependency injection using protocol is that it facilitates the unit-testing of client's functionalities as test dependencies can be used instead of production ones.

2.2.6 Unit testing

Unit testing is programming practice that separates the source code into small modular software unit and writes tests to those individual code unit. The main purpose of unit testing is to guarantee that a code unit functions correctly with different set of associ-

ated control data, usage procedures, and operating procedures [24]. Unit testing is traditionally a motivator for programmers to create testable, maintainable and decoupled code bodies.

As mentioned earlier, protocol-oriented design is an approach to avoid tightly coupled objects in software development and facilitate unit testing process. This can be obtained by defining the service or dependency type by adding abstraction layer instead of using a concrete type. The dependency injected to the client will then be the actual production-implemented object. Otherwise, mocked or faked objects are used in unit testing environment. The technique of decoupling the relationship between client and the dependency makes it easier to test each individual unit since they no longer depend on one another.

3 Practical implementation

3.1 Application

The application is created for the purpose of helping students to know which courses they need to take in order to be ready for certain kind of jobs. Each job has a minimum requirement of different skill sets and each course in school will provide students with specific skill sets. Courses also have pre-requirements, meaning that students are required to complete a specific set of courses or have specific skills in order to be eligible to enroll to. After finishing a course, student will gain a specific number of skills. Based on their current skills and the skills that their dream job requires; the application will calculate and provide them with different sets of minimum number of courses they need to complete.

3.2 Approach

The whole development process is divided into six different phases:

- Requirements
- Graphic design
- Software design

- Software development
- Testing
- Release

The development process follows Test-driven development (TDD) approach. TDD is a software development practice that promotes writing code only to pass certain tests [26]. First, programmer starts by writing test designed for a specific function, then writes bare minimum of code to fulfill the test. The new code will be then refactored until it meets a certain standard. In TDD, programmer always writes tests before the actual code. There are many advantages of using TDD approach for the software development. Not only it guarantees the correctness of the software, but it also helps programmers to think how the program should work and write minimal and optimal code. Writing test before actual implementation identifies the problem quickly and reduces the time spent on rework or on debugger in the future. A 2005 study found that programmers who apply TDD in their development process prove to write more tests, and writing more tests enhances their productivity [27].

When the program gets larger, maintaining the code base gets more difficult. With large amount of unit tests, it will help to spot where the problem is and how it affects the system quicker.

3.3 Technologies

The project is written completely in Swift 4.2 using XCode 10.1 as the Integrated Development Environment (IDE).
I use Core Data, the Cocoa-native persistence framework, to save user information and other related information on disk.

3.4 Application architecture

In the application, I use Model-View-View Model (MVVM) as the architectural design pattern. This design pattern was invented my Microsoft architects to remove data states from user interfaces. MVVM is designed to separate the concerns between the user interface logic and the business logic.

MVVM uses a technique called data binding which is used to create a communication between the view and view model. After the binding is correctly set up, when change is made to the data, it will reflect in the view automatically and correspondingly. For data binding, I am using RxSwift which is an open source library hosted in Github.

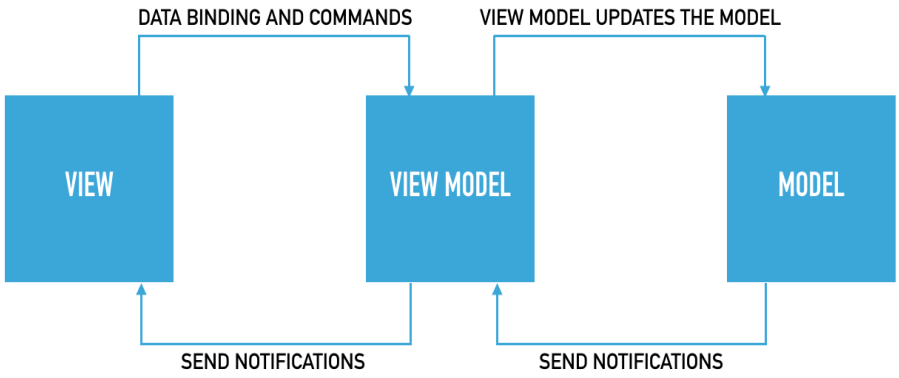

Figure 10. Interaction between components in MVVM.

3.4.1 Model-View-View Model

Model-View-View Model is an architectural design pattern. In this design pattern, Model, View and View model are the three main components. Each component has separate responsibilities and concerns related to how software is built.

The main difference between MVC and MVVM is that in MVVM, an extra component, view model, is introduced and how data is populated to views. View model is an object specifically designed for the view. It has properties that represent different states of the view and methods that implement the logic behind the view. The view model of MVVM is the middle layer between the View and the Model, meaning that its responsibility is to transform data objects from the model into other data objects which are then handed over the View to present. Thus, View model handles most business logic, from handling Model-related data to View's display logic.

Moreover, the other difference is that view controller is also considered as a view component in MVVM. View controller no longer owns model but instead asks view model

for the data needed to update its view. For that reason, many of its responsibilities are shifted to the view model, thus helps to avoid massive view controller problem when dealing with MVC.

3.4.2 RxSwift

RxSwift is the official Reactive Programming library written in Swift for iOS. It has many of its counterpart written in other languages as well, e.g. RxJava, RxJS, Rx.NET, RxClosure, etc. There are different versions of Reactive Extensions (Rx) in different languages but the main concept stays the same.

3.4.2.1 Concept

Rx enables easy composition of asynchronous operations and event or data streams.

It combines observer pattern and iterator pattern to allow data to be handled through data sequences. It also offers a wide range of operators which can be applied to each sequence or multiple sequences. and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O [24].

3.4.2.2 Observables

The core of Rx is observables. Observable represents a stream of data or asynchronous events which can be used to notify other objects of the changes of certain data source or event [24]. Observable is a wrapper of different observer techniques in Cocoa platform such as notifications, delegations, callbacks, etc. Without observables, handling data flow and data consistency in different places can be challenging since programmers have to integrate multiple chained callbacks or global notifications. It thereby reduces the readability and testability of the code and makes the software more prone to bugs.

3.5 Project structure

3.5.1 Models

There are four essential data models in the application project, which are *Profile*, *Course*, *Skill*, *Job*. Core Data is used to store all of these data information. The following figure shows how I design the models and the relationship between them.

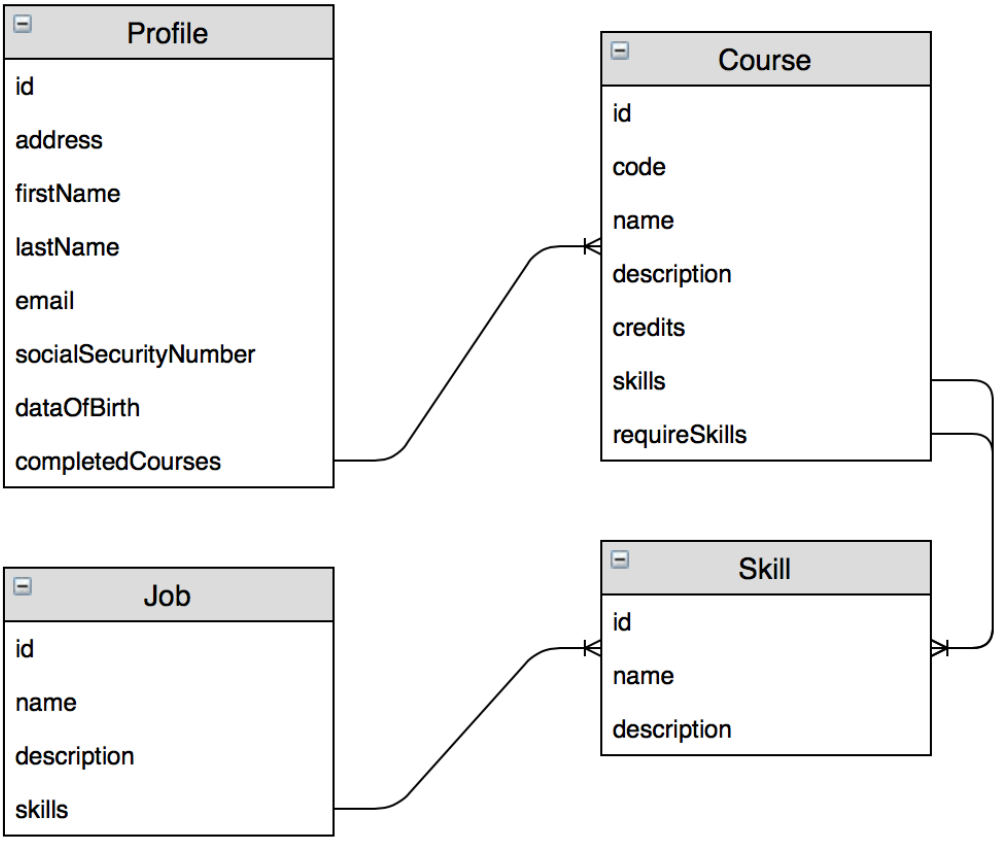

Figure 11. Relationship between models

In order to assist the work with Core Data when creating, retrieving, updating and deleting data of any model types, having a specific service for that is necessary. With POP, here I started with a protocol named `CoreDataStack` and added all required functionalities and properties that it must have.

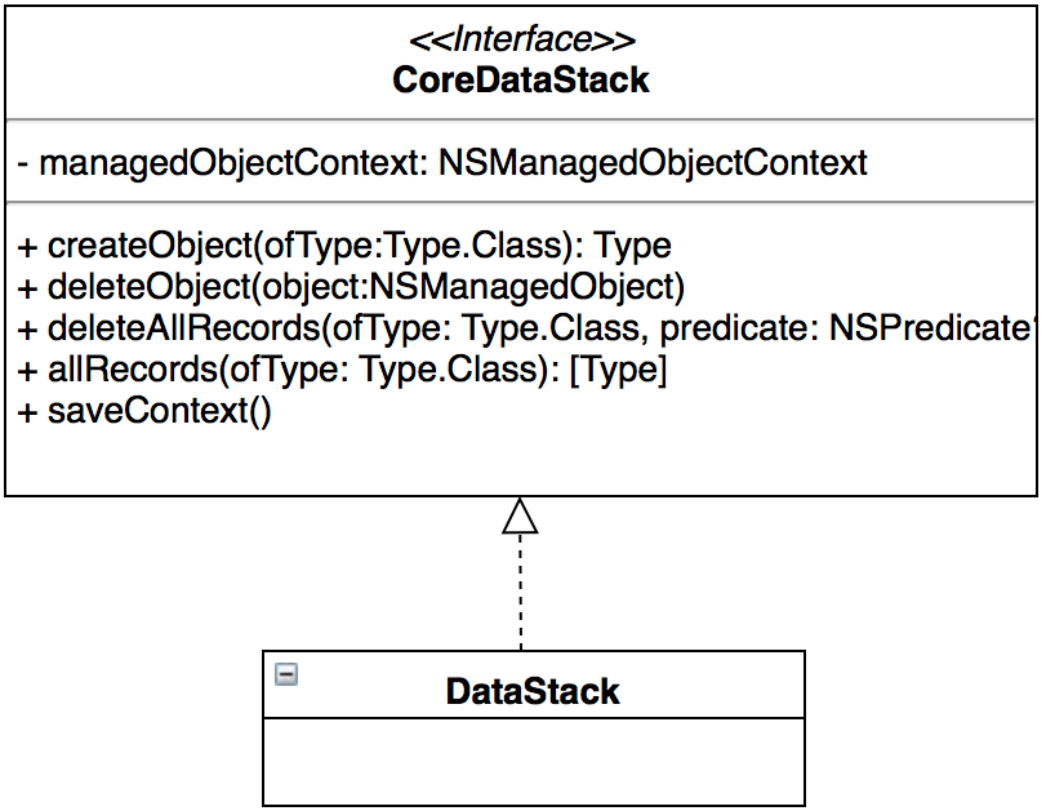

Figure 12. XML of DataStack.

```
protocol CoreDataStack {
    var managedObjectContext: NSManagedObjectContext { get }

    func createObject<Type: NSManagedObject>(ofType type: Type.Type) -> Type
    func deleteObject(_ object: NSManagedObject)
    func deleteAllRecords<Type: NSManagedObject>(ofType type: Type.Type, predicate: NSPredicate?)
    func allRecords<Type: NSManagedObject>(ofType type: Type.Type) -> [Type]
    func saveContext()
}
```

Figure 13. CoreDataStack protocol.

The use of generic types in the methods are important since it is not necessary to cast the type of the returning values. Thus, it improves the effectiveness of all `CoreDataStack` types.

By taking advantages of another powerful feature of protocol, protocol extension, I created a default implementation for `CoreDataStack` since there must not be any differences between different concrete types conforming it.

```
extension CoreDataStack {

    // Get fetch result controller
    func fetchResultController<ItemType: NSManagedObject>(type: ItemType.Type,
                                                          predicate: NSPredicate?,
                                                          sortDescriptors: [NSSortDescriptor]?,
                                                          sectionNameKeyPath: String?,
                                                          cacheName: String?) ->
                                                              NSFetchedResultsController<ItemType> {
        let request = fetchRequest(type: type, predicate: predicate, sortDescriptors: sortDescriptors)
        request.predicate = predicate

        if let sortDescriptors = sortDescriptors {
            request.sortDescriptors = sortDescriptors
        }

        return NSFetchedResultsController(fetchRequest: request, managedObjectContext:
            managedObjectContext, sectionNameKeyPath: sectionNameKeyPath, cacheName: cacheName)
    }

    private func fetchRequest<ItemType: NSManagedObject>(type: ItemType.Type,
                                                         predicate: NSPredicate?,
                                                         sortDescriptors: [NSSortDescriptor]?) ->
                                                             NSFetchRequest<ItemType> {
        let entityName = String(describing: ItemType.self)
        let request: NSFetchRequest<ItemType> = NSFetchRequest(entityName: entityName)
        request.predicate = predicate

        if let sortDescriptors = sortDescriptors {
            request.sortDescriptors = sortDescriptors
        }

        return request
    }
```

Figure 14. Default implementations of `CoreDataStack` protocol

Having the protocol ready is only the first step, I need to create a specific type which conforms the protocol and implements all the actual implementations. Thus, I created `DataStack` which is the only responsible class when working with Core Data. Even though there is only one type that conforms `CoreDataStack` in the whole project, it is beneficial to use the protocol type instead of tightly coupled `DataStack` type. The benefit is to enhance the testability and scalability. First of all, testability is improved when `CoreDataStack` is used as a service in a different class or type. In order to guarantee that all the functionalities and implementations of that Class is working properly, having the service as an instance of `DataStack` is not an optimal solution as it creates, deletes and modifies data directly from the database. Moreover, it does not provide with the flexibility to test all different real-life scenarios. The solution for that is having the service as a mock object from a type that conforms protocol `CoreDataStack`. The

mock class is also required to have to the required implementations, but I can do anything to make testing all different scenarios easier. Secondly, scalability or flexibility is acquired in case where actual implementations are updated in the future. In these cases, replacing the service with a proper type is all that requires. It is not necessary to update the functionalities of other parts of the software. In addition, since all the tests are independent from the actual implementation, everything continues to work without any modification. The following figure shows the actual implementation of `DataStack`.

```
class DataStack: CoreDataStack {

    static var shared: DataStack = DataStack()

    var managedObjectContext: NSManagedObjectContext {
        return persistentContainer.viewContext
    }

    private var viewContext: NSManagedObjectContext {
        return persistentContainer.viewContext
    }

    func createObject<Type: NSManagedObject>(ofType type: Type.Type) -> Type {
        return NSEntityDescription.insertNewObject(forEntityName: entityName(for: type), into:
            viewContext) as! Type
    }

    func deleteObject(_ object: NSManagedObject) {
        persistentContainer.viewContext.delete(object)
    }

    func deleteObjects(_ objects: [NSManagedObject]) {
        objects.forEach { persistentContainer.viewContext.delete($0) }
    }

    func allRecords<Type>(ofType type: Type.Type) -> [Type] where Type : NSManagedObject {
        let request = fetchRequest(ofType: type)

        do {
            let result = try viewContext.fetch(request)
            return result
        } catch {
            return []
        }
    }
```

Figure 15. Implementation of DataStack.

3.5.2 View models

In the ideal MVVM world, every view and view controller should have one and only one view model. The application has five main screens (view controller) and four other views that require its own view model. Therefore, there are total nine different view models in the project.

Following the same approach, I created a equivalent protocol for every view model. Let's first take a look at protocol `ProfileViewModelType`:

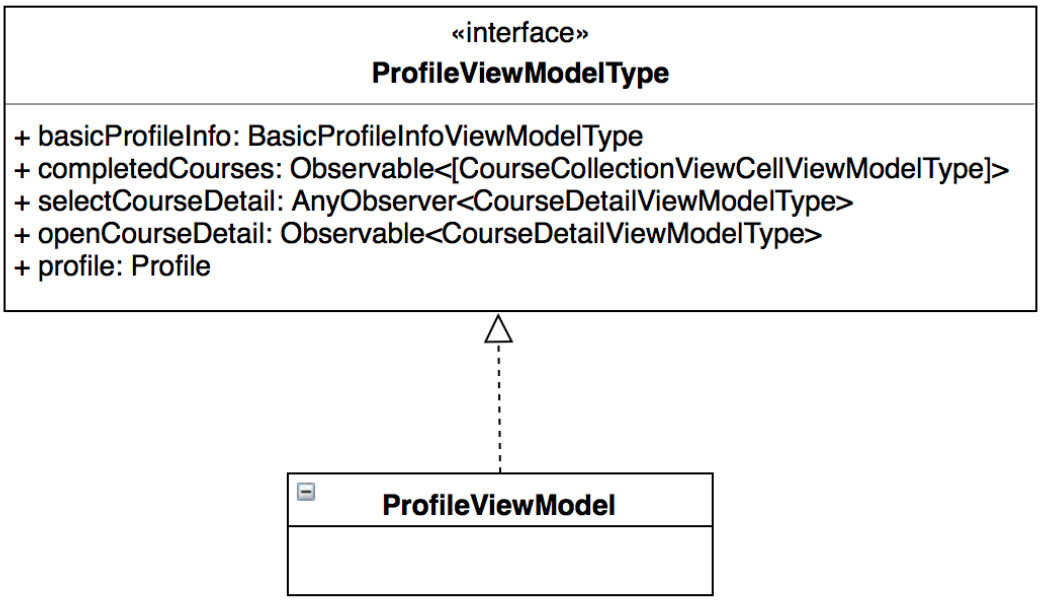

Figure 16. XML of ProfileViewModel

```
// MARK: ProfileViewModelType
protocol ProfileViewModelType: ProfileBasedViewModelType {
    var basicProfileInfo: BasicProfileInfoViewModelType { get }
    var completedCourses: Observable<[CourseCollectionViewCellViewModelType]> { get }

    var selectCourseDetail: AnyObserver<CourseDetailViewModelType> { get }
    var openCourseDetail: Observable<CourseDetailViewModelType> { get }

    var profile: Profile { get }
}
```

```
// MARK: ProfileViewModel
class ProfileViewModel: ProfileViewModelType {

    var profile: Profile

    var selectCourseDetail: AnyObserver<CourseDetailViewModelType>

    var openCourseDetail: Observable<CourseDetailViewModelType>

    var basicProfileInfo: BasicProfileInfoViewModelType

    var completedCourses: Observable<[CourseCollectionViewCellViewModelType]>

    required init(profile: Profile) {
        self.profile = profile
        basicProfileInfo = BasicProfileInfoViewModel(profile: profile)
        let completedCourses = profile.rx.completedCourses
        self.completedCourses = completedCourses
            .map {
                Array($0.map { CourseCollectionViewCellViewModel(course: $0, completedCourses:
                    completedCourses) })
                    .sorted { $0.name < $1.name }
            }

        let openCourseDetail = PublishSubject<CourseDetailViewModelType>()
        selectCourseDetail = openCourseDetail.asObserver()
        self.openCourseDetail = openCourseDetail.asObservable()
    }

}
```

Figure 17. Implementation of ProfileViewModel

In Figure 17, I make class `ProfileViewModel` conform protocol `ProfileViewModelType`. As mention earlier in this study, creating an abstraction layer of `ProfileViewModelType` is to hide irrelevant characteristics of the model object and only expose what is necessary to communicate with the other components in the software. All the complexity and detail are hidden inside `ProfileViewModel`. Furthermore, abstraction makes it easier for unit-testing and replacing one component of the software with completely different component. I already mentioned it in previous section and continue with more detail in section 3.5.3.

```
class ProfileViewModelTests: XCTestCase {
    var sut: ProfileViewModel!
    var profile: Profile!
    var dataStack: DataStack!
    var disposeBag: DisposeBag!
    var scheduler: TestScheduler!

    override func setUp() {
        // Put setup code here. This method is called before the invocation of each test method in the class.
        dataStack = DataStack()
        profile = dataStack.createObject(ofType: Profile.self)
        sut = ProfileViewModel(profile: profile)
        disposeBag = DisposeBag()
        scheduler = TestScheduler(initialClock: 0)
    }

    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each test method in the class.
        dataStack.deleteObject(profile)
        dataStack.deleteAllRecords(ofType: Course.self)
    }


    // Test completedCourses are properly updated after profile's completedCourses is updated
    func testUpdatingCompletedCourses() throws {
        try testCompletedCourses(equalTo: [])

        var set1 = Set<Course>()
        let course1 = dataStack.createCourse(id: "1234", code: "AAA", numberOfCredits: 5, name: "Course 1")
        let course2 = dataStack.createCourse(id: "2345", code: "BBB", numberOfCredits: 3, name: "Course 2")
        let course3 = dataStack.createCourse(id: "3456", code: "CCC", numberOfCredits: 4, name: "Course 3")
        set1.insert(course1)
        set1.insert(course2)
        set1.insert(course3)

        profile.completedCourses = set1 as NSSet
        let completedCourses = profile.rx.completedCourses

        try testCompletedCourses(equalTo: [CourseCollectionViewCellViewModel(course: course1, completedCourses: completedCourses),
                                           CourseCollectionViewCellViewModel(course: course2, completedCourses: completedCourses),
                                           CourseCollectionViewCellViewModel(course: course3, completedCourses: completedCourses)])


        let course4 = dataStack.createCourse(id: "2222", code: "DDD", numberOfCredits: 5, name: "Course 4")
        let course5 = dataStack.createCourse(id: "1111", code: "EEE", numberOfCredits: 15, name: "Course 5")
        var set2 = Set<Course>()
        set2.insert(course4)
        set2.insert(course5)
        profile.completedCourses = set2 as NSSet

        try testCompletedCourses(equalTo: [CourseCollectionViewCellViewModel(course: course4, completedCourses: completedCourses),
                                           CourseCollectionViewCellViewModel(course: course5, completedCourses: completedCourses)])
    }

    private func testCompletedCourses(equalTo courses: [CourseCollectionViewCellViewModel]) throws {
        if let completedCourses = try sut.completedCourses.toBlocking().first() as? [CourseCollectionViewCellViewModel] {
            expect(completedCourses) == courses
        }
    }
}
```

Figure 18. Tests for ProfileViewModel

3.5.3 ViewControllers

View controllers play an important role in all iOS applications. In MVVM architectural pattern, view controller acts similarly as a view with the main responsibilities are populating data to UI elements and driving user interaction to the view model via com-

mands. With the use of coordinators, view controllers even no longer manage the application flow. Since the responsibilities are compact and straightforward, thereby makes it easier to unit test the view controller.

As I mentioned above, `ProfileViewModelType` protocol has everything that can be used to bind to a `ProfileViewController` and commands that can be given from user interaction. `basicProfileInfo` and `completedCourses` represent all the data for the view controller. Those two properties are themselves view model and view model collection which provide data to the subviews living inside `ProfileViewController`. In this case, `ProfileViewModelType` is the parent of its child view models. On the other hand, I use `selectCourseDetail` and `openCourseDetail` as the command and output. `ProfileViewController` use the two properties in order to perform task when user selects a course, `selectCourseDetail` for firing the action and `openCourseDetail` for performing it. The following figure shows the actual implementation of `ProfileViewModel`.

```
class ProfileViewController: UICollectionViewController {

    private(set) var viewModel: ProfileViewModelType

    let sectionDataSource = RxCollectionViewSectionedReloadDataSource<ProfileViewSection>(
        configureCell: { dataSource, collectionView, indexPath, item in
            let cell = collectionView.dequeueReusableCell(withReuseIdentifier: cellReuseIdentifier, for:
                indexPath) as! CourseCollectionViewCell
            cell.populate(from: item)
            return cell
    })

    init(viewModel: ProfileViewModelType) {
        self.viewModel = viewModel
        super.init(nibName: "ProfileViewController", bundle: nil)
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    private let disposeBag = DisposeBag()

    override func viewDidLoad() {
        super.viewDidLoad()

        // Register cell classes
        collectionView!.register(UINib(nibName: "CourseCollectionViewCell", bundle: nil),
            forCellWithReuseIdentifier: cellReuseIdentifier)
        collectionView!.register(UINib(nibName: "BasicProfileInfoCollectionReusableView", bundle: nil),
            forSupplementaryViewOfKind: UICollectionView.elementKindSectionHeader, withReuseIdentifier:
            headerReuseIdentifier)

        // Do any additional setup after loading the view.
        title = "Profile"
        setUpBindings()
    }

}
```

Figure 19. Implementation of ProfileViewController.

Now `ProfileViewModel` can be used in `ProfileViewController` just like in Figure 19. Method `setupBindings()` does the proper data and action bindings between the view controller and its view model.

`ProfileViewController` has an abstraction layer for its own `viewModel` property and this is the up side of protocol-oriented approach. It creates a loose coupling between `ProfileViewController` and its `viewModel` dependency. Testing the functionalities of `ProfileViewController` and how it works with its own `viewModel` property can be carried out without having the actual implementation of `ProfileViewModel`. Creating a mocked, stubbed or faked instance is frequently the optimal solution in unit testing environment.

```
class ProfileViewModelStub: ProfileViewModelType {

    var basicProfileInfo: BasicProfileInfoViewModelType

    var completedCourses: Observable<[CourseCollectionViewCellViewModelType]>

    var selectCourseDetail: AnyObserver<CourseDetailViewModelType>

    var openCourseDetail: Observable<CourseDetailViewModelType>

    var profile: Profile

    private var completedCoursesObserver: AnyObserver<[CourseCollectionViewCellViewModelType]>

    required init(profile: Profile) {
        self.profile = profile
        basicProfileInfo = MockBasicProfileInfoViewModel(profile: profile)
        let completedCourses = PublishSubject<[CourseCollectionViewCellViewModelType]>()
        self.completedCourses = completedCourses.asObservable()
        completedCoursesObserver = completedCourses.asObserver()
        let openCourseDetail = PublishSubject<CourseDetailViewModelType>()
        selectCourseDetail = openCourseDetail.asObserver()
        self.openCourseDetail = openCourseDetail.asObservable()
    }

    func updateCompleteCourses(_ courses: [CourseCollectionViewCellViewModelType]) {
        completedCoursesObserver.onNext(courses)
    }
}
```

Figure 20. ProfileViewModel implementation for testing

Two differences between `ProfileViewModelStub` and `ProfileViewModel` are `basicProfileInfo` and `completeCourses` properties. While `basicProfileInfo` is another mocked instance, `completeCourses` behaves differently than its counterpart. In `ProfileViewModel`, `completeCourses` emits new value into the stream every time student adds or remove a course from its profile. On the other hand, the only way for `completeCourses` in `ProfileViewModelStub` to emit new value is to manually call method `updateCompleteCourses(_: [CourseCollectionViewCellViewModelType])`. By doing this, different test scenarios can be produced without having to update student information from the database. This approach facilitates and enhances the flexibility of unit-testing process.

After having the stub class and defining all the test cases for the `ProfileViewController` class, writing tests is straightforward and simple.

```
class ProfileViewControllerTests: XCTestCase {

    private var dataStack: DataStack!
    private var profile: Profile!
    private var viewModel: ProfileViewModelStub!
    private var sut: ProfileViewController!
    private var scheduler: TestScheduler!
    private var disposeBag: DisposeBag!

    override func setUp() {
        // Put setup code here. This method is called before the invocation of each test method in
            the class.
        disposeBag = DisposeBag()
        scheduler = TestScheduler(initialClock: 0)
        dataStack = DataStack.shared
        profile = dataStack.createObject(ofType: Profile.self)
        viewModel = ProfileViewModelStub(profile: profile)
        sut = ProfileViewController(viewModel: viewModel)
        _ = sut.view // load view
    }

    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each test method in
            the class.
        dataStack.deleteObject(profile)
    }

    // Test navigation title
    func testsViewControllerTitle() {
        expect(self.sut.title) == "Profile"
    }

    // Test populate data correctly
    func testCollectionViewHasCorrectNumberOfItems() {
        var courses = [FakeCourseCollectionViewCellViewModel(),
            FakeCourseCollectionViewCellViewModel()]
        viewModel.updateCompleteCourses(courses)
        expect(self.sut.collectionView.numberOfItems(inSection: 0)) == 2

        courses = [FakeCourseCollectionViewCellViewModel(), FakeCourseCollectionViewCellViewModel(),
            FakeCourseCollectionViewCellViewModel(), FakeCourseCollectionViewCellViewModel(),
            FakeCourseCollectionViewCellViewModel()]
        viewModel.updateCompleteCourses(courses)
        expect(self.sut.collectionView.numberOfItems(inSection: 0)) == 5
    }

    // Test correct supplementary view
    func testCollectionViewUsesCorrectHeaderViewType() {
        let courses = [FakeCourseCollectionViewCellViewModel()]
        viewModel.updateCompleteCourses(courses)
        expect(self.sut.collectionView.supplementaryView(forElementKind:
            UICollectionView.elementKindSectionHeader, at: IndexPath(row: 0, section: 0)) is
            BasicProfileInfoCollectionReusableView).to(beTrue())
    }

    // Test correct UICollectionViewCell class
    func testCollectionViewUsesCorrectCellType() {
        let courses = [FakeCourseCollectionViewCellViewModel(),
            FakeCourseCollectionViewCellViewModel()]
        viewModel.updateCompleteCourses(courses)
        expect(self.sut.collectionView.cellForItem(at: IndexPath(row: 1, section: 0)) is
            CourseCollectionViewCell).to(beTrue())
    }

}
```

Figure 21. Tests for ProfileViewController.

As shown in Figure 21, I used stubbed and faked objects to test the functionality of `ProfileViewController`. This technique makes sure that the behavior of the system under test is independent from the real-world implementation of its dependencies, making it easier to test and still guaranteeing that the program works as expected. Moreover, for more complex test cases, a more versatile mock, stub or fake can recreate different test cases easily and efficiently.

4 Results

As a esult of this study, an app called DreamJob was created. DreamJob is only a demo application and it is not available for download from any sources. The main functionality of the application is that it helps students to plan their study based on their career interests.

When a client, most often a student, opens the application for the first time, the application requires them to input their information. After that, they will be brought to the main views which show a list of predefined available courses and jobs. Students can add courses that they already completed to their profile. In addition to that, student can select the job that they are interested in. Based on their completed courses and interested job, the application does a searching algorithm to show them all the options for study path that they must take to get all skills required by the job.

The application is finished with the code base which is open for extension. Currently, the application does not support courses and jobs to be fetched remotely. That feature, however, could be added with ease without much modifications but additions.

5 Discussion

5.1 Study outcome

From functionality wise, the application is not ready to release to App Store as it is not meant to be. However, it could be extended for more production features thanks to its current design. The project which follows POP paradigm possesses some major differences from most of current Cocoa projects. With POP, it improves the testability and

extensibility of the project by decoupling its components. Each component of the software can be replaced by another one with ease, either between testing and production environments or between different configurations.

In addition, the project is also following MVVM architectural design pattern instead of traditional MVC. This adds an additional layer to different levels of components in the software but creates a better separation of concerns. Thus, the software has view model objects which takes away many of the view controllers' responsibilities. Another tool, used in combination with MVVM, is RxSwift which makes handling data-binding between view models and views, asynchronous processes easily and productively. MVVM and RxSwift are a great combination; nevertheless, there is no silver bullet when it comes to application architecture. Since MVVM shifts many responsibilities toward the view models; with more complex view or view controller, view model can become massive and not reusable at some point. Moreover, not all Cocoa programmers are familiar with MVVM and Reactive Extensions. Especially with Rx, it is a difficult concept, thus getting new team members onboard takes longer and more effort.

Finally, the project was written following TDD approach which means the code is written to pass certain test cases. This habit of writing code ensures that the program functions as expected. It also adds a protection layer to the code written so that any modification in the future will not break the current behavior of the application. TDD is even more beneficial when multiple programmers are working on the same project and updates can happen at any given time.

5.2 Limitations

As mentioned in the study, POP is not only about using protocols, but about using value types over reference types also. However, the project does not show the application of POP at full potential. Due to the nature of RxSwift and Core Data, most part of the program was implemented with class instead of structure. RxSwift's observables and Core Data's objects are both reference types. Instances that mostly work with observables or Core Data objects are defined as reference types are as value type has little to no advantages to their counterpart. When making value types containing multiple other reference types, we still have to deal with problems when it comes to using objects such as thread-safety, race conditions, implicit sharing, etc. This is why I implemented types that are made up by reference types as reference type.

In addition, the usage of RxSwift, a third-party library, makes the project strongly dependent on RxSwift's APIs and code syntaxes for asynchronous processes and data binding. Replacing the current implementation from RxSwift with a different mechanism takes an enormous amount of work. Moreover, the concept of Reactive Extensions is unfamiliar to a majority of Cocoa programmers. Therefore, projects using RxSwift result in a steep learning curve to new comers. The big difference in coding paradigm makes it more challenging to program, test and debug the software.

5.3 MVVM vs MVC

I already mentioned the advantages that MVVM has over MVC. The additional component, view model, in the architectural design pattern separates many of responsibilities from view controllers which programmers often find problematic when working it MVC. View models also improves the readability of the code as it is the object that is specifically designed for the view and works directly with the models.

Secondly, another additional component, coordinator, takes away the navigation responsibility from view controllers. Coordinators are using the navigation concept, thus have the responsibility to control the flow of the application and navigate between different screens. With coordinators, view controllers take this responsibility, thus it makes them less reusable and testable.

6 Conclusion

In the programming community, scientists and programmers have never stopped inventing and creating new methods to make better software. For Cocoa programmers specifically, many different architectural design patterns have been introduced during the last decades. Some of them can be listed such as MVC, MVVM, Model-View-Presenter (MVP), VIPER [1]. And, until WWDC 15 with the introduction of POP, Cocoa programmers have had a brand-new additional approach to develop software.

POP is programming paradigm to create reusable and testable software components. Owning various powerful features such as protocol extension, protocol inheritance, associated types, constraints, generics, etc., Swift itself and Swift protocols open up many different programming techniques. Moreover, POP also encourages the usage of value

types over reference types. Value types, while being optimized by Swift compiler, produces higher performance than their counterpart in many parts of the software and eliminates many drawbacks that OOP and classes bring up such as implicit sharing, race conditions, lost type relationship, thread safety, etc. Value types are passed by value, so updating a value type does not reflect the change somewhere else. Thus, it is safer when handling different states than it is with objects.

In addition, using protocol as abstraction removes tight coupling between different components; thus, makes it easier to test and maintain the code base. By defining the dependencies in the form of protocol, we can easily reuse objects in different situations even though they might require differently. For example, mocked implementation of protocol can be used to facilitate unit testing process. However, applying POP is not only for better code quality but also for building better software and improving software performance.

In conclusion, POP and MVVM are just different tools for building software. They are not bulletproof to be used anywhere in the development process. In fact, they should be used when it is more beneficial and applicable. Swift is also an OOP language and many of Cocoa frameworks are built upon classes and inheritance. Therefore, object-oriented design still plays an important role in Cocoa project. Using OOP, or POP, or both where appropriate will improve much of code reusability and quality. Similarly, in cases where observables and data bindings are not applied, MVC might come as a better option than MVVM.

References

1 iOS architecture patterns: A guide for developers. Available from: https://thinkmobiles.com/blog/ios-architecture-patterns/ [cited April 13, 2019]

2 Apple. Swift Has Reached 1.0. Available from: https://developer.apple.com/swift/blog/?id=14 [cited April 6, 2019]

3 Model-View-Controller. Available from: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html [cited April 22, 2019]

4 Kindler, E.; Krivy, I. (2011). "Object-Oriented Simulation of systems with sophisticated control". International Journal of General Systems: 313–343.

5 What Is Object-Oriented Programming & Why Is It Important?. Available from: https://www.upwork.com/hiring/development/object-oriented-programming/ [cited April 22, 2019]

6 Gamma; Helm; Johnson; Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. (Bruce 2002, 2.1 Objects, classes, and object types).

7 Value types and reference types. Available from: https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/data-types/value-types-and-reference-types [cited April 6, 2019]

8 Value and reference types. Available from: https://developer.apple.com/swift/blog/?id=10 [cited April 6, 2019].

9 Swift vs. Objective-C: A Look at iOS Programming Language. Available from: https://www.upwork.com/hiring/mobile/swift-vs-objective-c-a-look-at-ios-programming-languages/ [cited April 22, 2019]

10 Features of Object-Oriented Programming (OOP). Available from: http://studytipsandtricks.blogspot.com/2012/04/features-of-object-oriented-programming.html?m=1 [cited April 6, 2019].

11 Rogers, Wm. Paul (18 May 2001). "Encapsulation is not information hiding". JavaWorld.

12 Johnson, Ralph (August 26, 1991). "Designing Reusable Classes" (PDF). www.cse.msu.edu.

13 OOP Concept for Beginners: What is Abstraction? Available from: https://stackify.com/oop-concept-abstraction/ [cited April 13, 2019].

14 Protocol-Oriented Programming in Swift. Available from: https://developer.apple.com/videos/play/wwdc2015/408/ [cited April 13, 2019].

15 What is defensive copying? Available from: http://www.javacreed.com/what-is-defensive-copying/ [cited April 14, 2019].

16 Differences between Stack and Heap. Available from: http://net-informations.com/faq/net/stack-heap.htm [cited April 13, 2019].

17 Inheritance. Available from: https://docs.swift.org/swift-book/LanguageGuide/Inheritance.html [cited April 22, 2019]

18 Protocols. Available from: https://docs.swift.org/swift-book/LanguageGuide/Protocols.html [cited April 13, 2019]

19 Generics. Available from: https://docs.swift.org/swift-book/LanguageGuide/Generics.html [cited April 13, 2019]

20 Gamma, Erich; Richard Helm; Ralph Johnson; John M. Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. p. 395.

21 Delegation. Available from: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html [cited April 13, 2019]

22 Dependency Injection Demystified. Available from: https://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html [cited April 13, 2019]

23 Inversion of Control Containers and the Dependency Injection pattern. Available from: https://www.martinfowler.com/articles/injection.html#FormsOfDependencyInjection [cited April 13, 2019]

24 Automated Defect Prevention: Best Practices in Software Management. Available from: https://www.wiley.com/en-us/Automated+Defect+Prevention%3A+Best+Practices+in+Software+Management-p-9780470042120 [cited April 13, 2019]

25 ReactiveX. Available from: http://reactivex.io/intro.html [cited April 13, 2019]

26 On the Effectiveness of Test-first Approach to Programming. Available from: https://nrc-publications.canada.ca/nparc/eng/view/object/?id=0420df64-f474-4072-8df6-c7b87c0de643 [cited April 13, 2019]

27 On the Effectiveness of Test-first Approach to Programming. Available from: https://nrc-publications.canada.ca/nparc/eng/view/object/?id=0420df64-f474-4072-8df6-c7b87c0de643 [cited April 13, 2019]