UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Comparative analysis of modern iOS architectures in different development stages

Author:
Andrii KOVAL

Supervisor:
Roxana MARKHYVKA

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2021

Declaration of Authorship

“The secret of success is to do the common thing uncommonly well.”

John D. Rockefeller Jr.

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Comparative analysis of modern iOS architectures in different development stages

by Andrii KOVAL

Abstract

Modern iOS development became much more versatile than it was several years ago. There are lots of different frameworks, approaches, patterns and architecture which can be used during the application development. In this thesis I aim to analyze most popular architectures for iOS development, compare them with each other and find out their benefits for different application and development stages.

Acknowledgements

Thanks to my supervisor Roxana Markhyvka who helped me with deriving thesis's subject and inspired me to write this work about iOS architectures.

I am grateful to my friends Hermann Yavoskyi and Volodymyr Chernetskyi for supporting each other along this way.

Contents

List of Figures

Abbreviations and Definitions

MVC	**M**odel **V**iew **C**ontroller
MVP	**M**odel **V**iew **P**resenter
MVVM	**M**odel **V**iew **V**iewModel
MV	**M**odel **V**iew
OOP	**O**bject **O**riented **P**rogramming
PoC	**P**roof **o**f **C**oncept
(G)UI	(**G**raphic) **U**ser **I**nterface
UX	**U**ser E**x**perience
API	**A**pplication **P**rogramming **I**nterface
KVO	**K**ey **V**alue **O**bserving
MVP	**M**inimum **V**iable **P**roduct
FRP	**F**unctional **R**eactive **P**rogramming
VIPER	**V**iew **I**nteractor **P**resenter **E**ntity **R**outing
UIViewController	Main Cocoa class to display separate screen.
Closure	Inline function type in Swift Programming Language
RxSwift/RxCocoa	3rd party Swift framework for reactive programming.

Dedicated to my Father, in loving memory…

Chapter 1

Introduction

1.1 Motivation

Information technology sphere develops extremely fast. Lots of technologies are used to create various software in different domains and for specific purposes.

People everyday use their phones to communicate, search for information, watch videos, play games, take photos and others. Mobile development companies create applications with extreme pace. But lots of them don't even have a chance to outshine the giant rivals, who possess millions of users.

Since the iOS development also is growing very fast as well, there are lots of new and old approaches for application development, but still in teams, who start project the first question before development stands which architecture to choose for the application and why (HARUN, 2019). Lately, the most popular modern architectures such as MVC, MVP, MVVM, VIPER, Clean Swift have captured the market and I want to compare them and to make the future decision on architecture a little bit more clear and grounded.

1.2 Problems

While choosing the application architecture, developers often face several challenges. First of all, which technologies are being expected to use, how often they will modify code due to numerous change requests, scale of the application, its design, etc.

Decisions on right architecture can solve all of these problems just because an experienced developer can predict some risks and make the right choice.

More serious problem nowadays is the low competence level of new developers compared to ones several years ago. This leads to bad quality code, a lot of useless refactoring, plenty of problems because of wrong architecture choice.

1.3 Goals

The goal of this thesis work is to analyze, compare and conclude which architecture will be a good choice for specific projects. As a result, given some conditions on the project we will be able to assume which architecture will suit this application best and why.

1.4 Background information

1.4.1 UIKit

Modern iOS development with Swift includes working with business and application logic. To implement second one the UIKit standard library provides developers with comprehensive interface and lots of UI/UX components to use (UIViewController, UIView, UINavigationController, etc).

1.4.2 OOP and SOLID

Every single mobile application is developed based on object oriented programming rules and following SOLID principles. Modern architectures help to develop applications and during this process developers can think less about possible OOP violations since these architectures and patterns were created exactly for avoiding such violations.

Chapter 2

Model-View architectures

2.1 Model View Controller

When getting started into iOS development, the first common architecture that someone finds out is Model-View-Controller, also known as Apple's MVC. This design pattern is quite simple, since it divides all the objects in your application into the three main types and determines the way they communicate with each other. These are Model, View Controller and let's look separately at each of them:

- Model - these objects are responsible for representing, processing and manipulating all the data within the iOS application. The most popular model-type objects are application data models that define the types, which are used inside the application. In addition to them, model type can include objects which can process, load, synchronize, save and perform other operations with the different data throughout the application.
- View - objects that are used to show the UI of the application. These objects know how to draw themselves and react to different events such as taps, handles, swipes and other user interactions. Main view-type objects are views and screens, which are visible to the user and they have the role of presenting data and giving the possibility to somehow edit this data.
- Controller - the objects of controller type serve as a bridge between the model and the view. As we will see further, since views cannot directly communicate with models, the controller provides this connection, coordinates the application flow and notifies view and model about changes, which should have any impact on data or UI.

Talking about dependencies in MVC, everything is quite intuitive. Controllers coordinate the application flows and provide communication between View and Model.

The key feature of MVC architecture is simplicity. Mostly it serves good for few-screens applications, pet projects, PoC's, etc. These applications do not include any complicated business logic and user flows, so all the obvious MVC's disadvantages are not so noticeable in the scope.

Why is MVC bad for enterprise applications, complex startup applications or applications with lots of screens, flows and business logic? This architecture cannot provide flexibility and reusability for big-scale projects and this will cause an extreme mess in the development process. As a result, the most famous problem of MVC will come to these projects - enormous Controllers, which violate OOP rules, especially single responsibility from SOLID principles (Dobrean and Diosan, 2019).

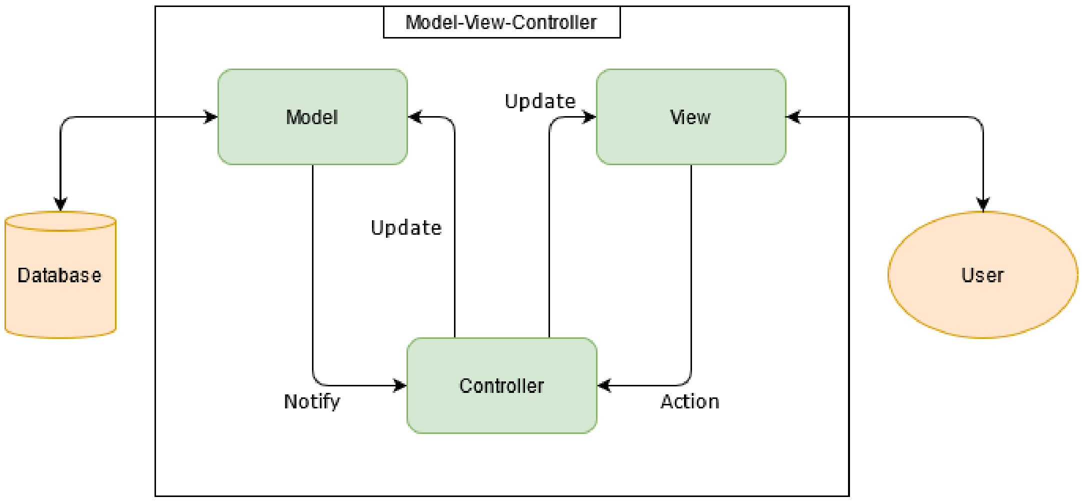

FIGURE 2.1: Model-View-Controller architecture.

2.2 Model View Presenter

Talking about MV-architectures I can't forget the MVP which stands for Model-View-Presenter. This architecture is quite similar to MVC, but has clear differences and solves some obvious MVC problems. MVP is a more scalable and flexible architecture, therefore it is more usable in real projects, than Apple's MVC.

The Model and View roles remain almost the same as in MVC with some changes, but now we get a new type, called Presenter. This object simplifies the development process and gives more possibilities to create complex user flows and GUI.

- Presenter - serves as a mediator between Model and View. At first sight it is the same as Controller, but actually it executes only a part of a MVC controller's work. Presenter is responsible for presentation logic in general, converting business data into readable UI format and vice versa. This object knows all about what should be shown on the screen and when the application should show specific information based on presentation conditions, implemented in Presenter. Moreover, it is responsible for providing communication between View and Model. Presenter handles user interactions received from View, can validate and process the data and also it reacts to changes in Model and forces the View to update accordingly.

As we can see, the Presenter is not the same object as the Controller. It is responsible for more data processing and presentation logic, communicates with model and view and handles the whole process of user interaction.

MVP is a little more complex than MVC and usually it uses more roles or layers to simplify the UIViewController or Presenter logic. For example, the common practice is to separate some data loading logic into an isolated layer, called Service. In this way, the code will be cleaner and development quality will be higher, since we isolate layers with different responsibilities, so that we are confident that Model knows nothing directly about user interaction on View and View doesn't need to know how the Model changes, because Presenter will handle both of these.

Model-View-Presenter can perfectly serve small and middle-scale applications, such as growing startups or mass market applications with several functionalities. Its moderate versatility makes it comfortable to scale such applications. Also, MVP

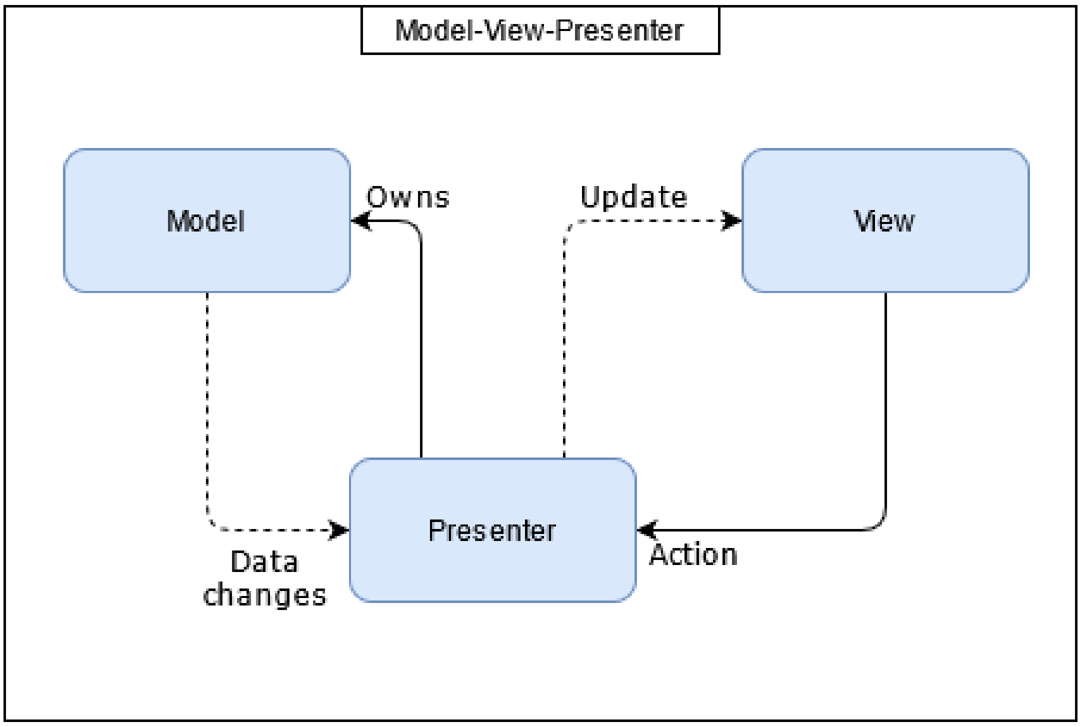

FIGURE 2.2: Model-View-Presenter architecture.

is flexible enough to work in Agile projects. Because of its layer separations you can without difficulties make changes into one of them and others won't be affected.

Since the MVP remains the MV-architecture it possesses some drawbacks. First of all, it is still not flexible enough for large enterprise applications with lots of screens. Moreover, it is not very comfortable to use with applications, where you want to implement some complex and custom UI components, since this can make your presenter overloaded.

2.2.1 MVP vs MVC

MVP and MVC are quite similar, because they both are MV-architectures and here are the same features of both architectures:

- If you don't want to use any other layers for data loading, you can still make the model responsible for this. All the fetching data from API, database or wherever can be implemented within the model layer and the architecture will still remain the correct MVP.
- Like all the MV-architectures, MVP suffers from unclear navigation. MVC and MVP don't provide us with proposed navigation layers, therefore this part is usually up to the developer.

Despite the similarity, there are some explicit differences between them.

- One of the main key differences is that the UIViewController is now strictly the part of the View layer. This solves the main MVC's problem with enormous Controllers, since now the Controller is spread to UIViewController and Presenter, which make the architecture be more SOLID

- View is not responsible for any work, except for presentation. Presenter is responsible for handling all the user interaction
- View is not responsible for any work, except for presentation. Presenter is responsible for handling all the user interaction

2.3 Model View ViewModel

The last architecture from the MV group is Model-View-ViewModel and this one probably is the most popular architecture for new projects recently. MVVM is a bit different from MVC and MVP with the way how the objects communicate with each other inside the application. Model and View stand for the same as in MVC and MVP, but in MVVM we have a new role called ViewModel (Luong Nguyen, 2017)).

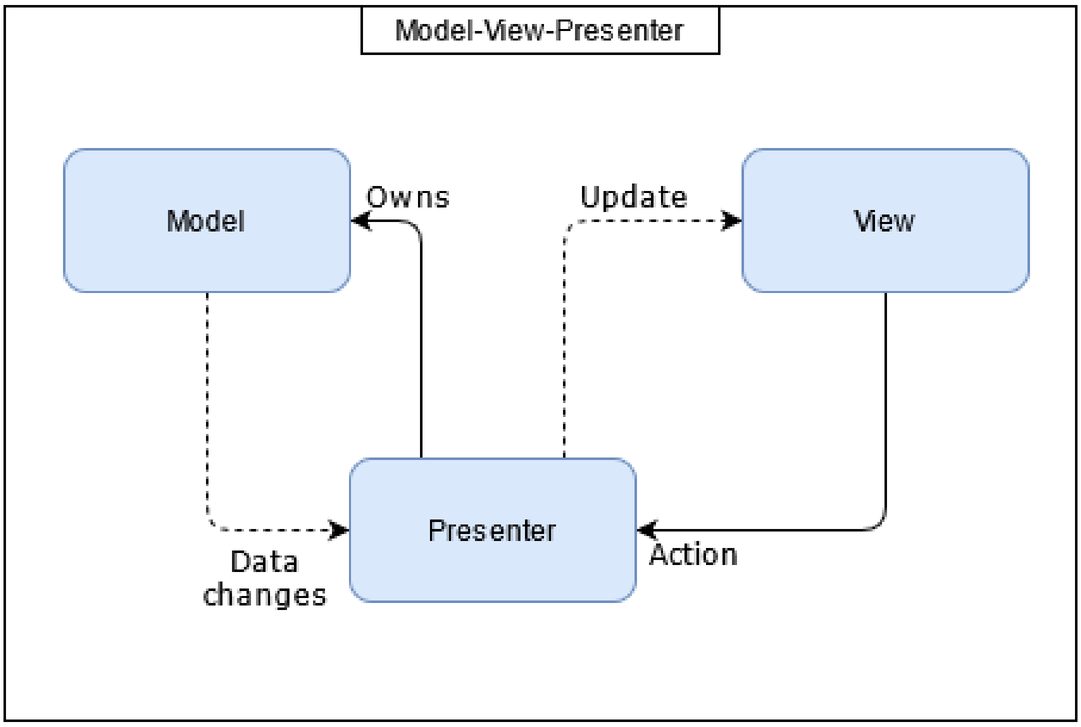

FIGURE 2.3: Model-View-Presenter architecture.

- ViewModel - simple object, which converts data into format to represent on View. VM is also responsible for controlling the state of View and handling interaction. The main difference between MVVM and previous MV-architectures is that the ViewModel communicates with View using data binding.

Data binding - is the way you can tie up several of your application objects to represent one state on different application layers. In MVVM it means that you can tie up your concrete View objects with corresponding data and whenever the data changes, the UI also changes.

There are different ways of data binding in iOS development, which can be used in MVVM:

- Closure binding - ViewModel when initialized in a View gets a closure, where all the binded objects are updated according to the corresponding data from

ViewModel. At the same time, this ViewModel has data fields with observers, which triggers on data changes and the closure is being performed. As a result, when the ViewModel gets any data updates from the Model layer, the closure is triggered and binded View objects update their state.

- KVO binding - more complex iOS implementation of data binding which is quite rarely used, since MVVM very often includes usage of FRP frameworks, such as a Rx, in case of iOS - RxSwift and RxCocoa.

  Key-value observing means that in a UIViewController you create observers on ViewModel data fields and when the specific data is changed, a concrete observer triggers and forces View to update a specific component.

- One of the most modern, popular and simplest ways to implement data binding is using the FRP framework, for example RxSwift and RxCocoa. The logic of this kind of data binding is the same as KVO, since underneath it works using Swift's key-value observing. However it is much more convenient to use and provides modern interfaces to effectively create reactive chains, which can perform not only data binding, but also do different operations with data, for example, filtering, sorting, mapping, reducing, etc.

Considering the above, ViewModel is a powerful pattern, which makes mobile developers use MVVM widely in their projects. ViewModel gives the opportunity to hold the state of each View component separately as well as the entire View screen. Data binding is an extremely convenient way to solve the problem of communication between View and Model. In addition, it helps the developer in writing testable code.

As for dependencies in MVVM, usually View holds the ViewModel and VM has weak reference to View to update the UI on trigger. ViewModel holds the Model and reacts on its changes, updating its own properties.

MVVM quite often is implemented without data binding and ViewModel is responsible for presentation logic and communication between View and Model. This can be quite confusing, but such MVVM implementation is not correct, since the actual pattern is MVP, but with a Presenter named as ViewModel. In modern iOS development this is a common problem among people who can't say the key distinctions between MVP and MVVM. However, as I've shown earlier, MVVM has a completely different concept of Model and View communication, which makes it a unique pattern.

Modern applications often include a lot of custom UI components and fancy animations. MVVM implementation is perfect for such applications, because you always have control of the UI state from ViewModel and can implement animations and different UI states quite easily in comparison to other architectures, which will require much more work and attention to details.

MVVM is a good pattern in different stages of application development, but it requires more skilled developers to work with. It also provides you with the opportunity to write very testable code and for big companies and big projects that is a significant advantage. This architecture is convenient for small, middle-scale applications and sometimes for large ones, which won't change significantly. The reason is that MVVM as all the MV architectures suffers from unclear navigation implementations and its modules mostly are badly reusable for different purposes. MVVM pattern gives a possibility to develop modern animatable GUI in the short term and can also serve perfectly for fast-growing startups, which need catchy applications to attract investors and users.

As well as other MV architectures, MVVM is not a very good choice for enormous enterprise software, since it has only three layers and in large applications this will result in a lot of messy unreadable and unsupportable code.

2.3.1 MVVM vs MVC

Even though MVC and MVVM are both MV architectures, they have some distinct differences, which make them serve for various purposes.

- To start with, the way data is processed and shown in application is extremely different. The MVC View layer does all the presenting work, whereas in MVVM there is Model, which processes and prepares data and ViewModel, which is responsible for data binding with the View layer. The MVVM approach is more complex but at the same time more powerful, since it allows one to manage the state of the View layer through the Model layer without direct connection between them. MVC architecture uses Controllers to establish communication View and Model and suffers from a problem of extremely large UIViewControllers (Aljamea and Alkandari, 2018).
- MVVM is more comfortable to use when developing applications with lots of customization, for example custom UI components or animations, because all the View objects bind to ViewModel's data properties and it is possible to easily trigger such animations, change data while animating, etc. MVC, counterwise, is less difficult in implementing small applications without specific components and this gives you the opportunity to write less code and get to the PoC or MVC (minimum viable product) faster.

As for similar features of both architectures:

- MVVM and MVC have one common problem of all MV architectures - navigation. When starting a project and using one of any MV architectures there is also a dilemma, which navigation implementation to choose for the application.
- Both these patterns do not need implementation of lots of layers, so they can be used for fast-pace application development or adding new features to applications in the shortest possible terms

2.3.2 MVVM vs MVP

As I said earlier, lots of iOS developers, who I worked with, quite often mistakenly write their code with the ViewModel layer thinking they use the MVVM pattern. But actually, the architecture they use is simple MVP, so let's consider similar features and find out why these patterns confuse developers.

- The concepts of ViewModel and Presenter are very similar with one distinction, which I will describe below. They both are responsible for providing ready-to-present data to View, handling user interaction, supplying communication for View and the Model.
- MVP and MVVM also inherit all the MV architectures problems (navigation, small amount of layers)

Despite the similarities of MVVM and MVP, lots of experienced developers continue to prefer using the MVVM in their applications and MVP is a much rarer choice.

- The key difference between MVVM and MVP is the difference in ViewModel and Presenter implementations. Presenter does the data filtering, processing and other operations itself, prepares data and passes it to View. Actually this object does everything needed for presentation of UI components except for rendering. In MVVM the data is being processed in the Model layer and ViewModel with its data binding serves as state manager for View components. The dependencies in MVVM are more solid, since the ViewModel instantly reacts to data changes and updates the View, whereas in MVP Presenter should react on user interaction, process data and then tell the View layer to update the UI.
- Since MVVM uses a more complicated data flow approach it requires more device resources and precise coding, while MVP does not use any unusual ways and developing new modules, modifying existing code is extremely straightforward.

2.4 Model-View architectures + Coordinator

Above I described and compared three most popular MV architectures and stated that all of them have a common problem with navigation within the application. Here I want to show the proposed navigation solution for these architectures, which works well and fits perfectly to each of the discussed architectures.

Defining any of these architectures as MV architecture we will add one more layer to the application called Coordinator. These objects can be on different application levels, depending on project size, amount of user flows and feature modules.

2.4.1 What is Coordinator?

In iOS development for any MV architecture it is very important to define specific navigation ways, because they do not manage application flows themselves. The good solution here is to add a new object, that knows nothing about data or UI, but knows all the possible application flows and navigation, for example which screen should be shown after Log In screen, or which flow should start, when we tap the specific button. Coordinator is the object we are looking for in this case.

Usually, UIViewController holds the Coordinator object and tells it to perform necessary navigation based on the user interaction. With this approach we move the navigation logic into a separate layer, which covers all application flows. There are different modifications of using Coordinators, but mostly there is one root AppCoordinator for the entire application which manages child Coordinators. Those child Coordinators at the same time can manage their own child coordinators, this is needed, when your flows are extremely large and one coordinator would have too many responsibilities. And lastly, the Coordinators define all the possible routings within their flows to cover user flows.

Common example: AppCoordinator holds OnboardingCoordinator, which is responsible for navigation inside the Onboarding flow: Log In, Sign Up, Password Recovery and other flows.

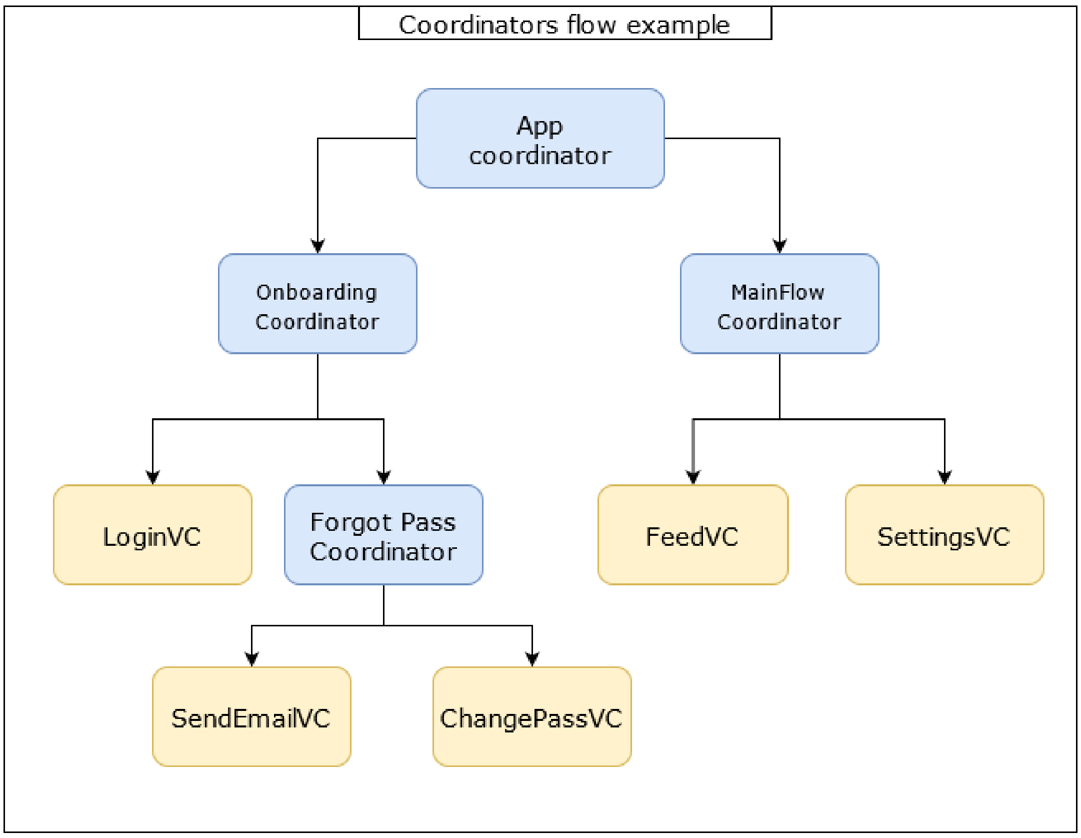

FIGURE 2.4: Example of several Coordinators in the application.

2.5 Conclusion

Summing up all the Model-View architectures, I made this short overview of each pattern to recall their distinctive features.

- Model-View-Controller - also known as Apple's MVC. Three explicit layers which have their responsibilities: Model - data operations, View - UI representation, Controller - mediator for Model and View.

  Convenient architecture for different small applications, for example, pet projects, startups or applications with simple core functionality. Using this pattern, developers often face the problem of large UIViewControllers, so that the code becomes hard to read, modify, refactor and debug.

- Model-View-Presenter - improved modification of MVC, where View inside can be separated into View and Controller and also has new layer Presenter, which is responsible for communication between Model and View, preparing data to show on UI and handle all the presentation logic within the screen.

  More advanced architecture than MVC, since each layer has its own special responsibility, but still this architecture is good for small and middle-scale applications due to bad reusability.

- Model-View-ViewModel - extremely interesting MV modification since most implementations require FRP to provide data binding within View and ViewModel. VM - new layer, which binds data and handles all user interaction, but no need to notify View about updates, since it can update reactively triggering on ViewModel data changes.

Convenient architecture for developing applications with extraordinary GUI and animations. In addition to this, MVVM implementations are good for testing in comparison to MVC and MVP. Can be used effectively to develop projects and features of different sizes, but some challenges may appear, when working on complicated flows.

All the MV architectures are quite compact, so each of them is perfectly maintainable in different development stages. Moreover, each of them is simple to understand and it helps in extending applications with new features.

Personally, for me, if developers are skilled enough and know how to use FRP, MVVM is the best choice from these patterns to use while developing applications. It provides the most testable code and efficient data management for creating projects of different sizes and domains.

Chapter 3

More complex architectures

3.1 VIPER

After we have seen the Model-View architectures we can dive into more advanced software patterns in the scope of iOS development. VIPER is an backronym for View-Interactor-Presenter-Entity-Routing (Adibowo, 2020) and with some modifications often defined as VIPER Clean Architecture. What is this, we I will explain in the next section and for now VIPER is a huge and massive architecture with precisely separated application layers where each layer has only one responsibility and this helps to write clean, maintainable code, which is quite easy to test.

Avoiding the Clean Architecture paradigm for now, VIPER is extremely efficient architecture, since its logical structure consists of distinct layers. With such features it is easy to isolate dependencies and solve the problem of MV architecture, especially MVC - Massive View Controller.

- View - renders UI and represents anything that is provided by the Presenter object. This layer is passive and does any application work except for rendering views. This layer is very similar to View in MV architecture.

- Interactor - interesting layer, which contains all the business logic of the application. These objects are responsible for manipulating data, performing different operations on it and this entire work is done without notifying the UI. So, the main task of Interactors is solving specific use cases using data models.

- Presenter - this layer is responsible for representation logic of UI. It communicates with Interactor and View and after Interactor performs some data operations, Presenter is ready to make the data ready for presentation for the user. Interesting fact, that Presenter usually does not work with Entities (Model objects). The object gets the data from Interactor in specific format and prepares it for View.

- Entities - simply the model objects in the application. These objects are used by Interactor to perform some business logic tasks and then in custom format transferred to Presenter. As I said earlier, Presenter doesn't know how to work with Entity objects.

- Routing - the navigation layer of the application. Navigation tasks consist of 2 parts - user interaction (trigger) and actual transition. First part is handled by Presenter, which processes all the user interaction and notifies that transition should be done. New defined object called Wireframe handles the second part of the navigation task. It usually possesses UINavigationController, UIWindow or other iOS standard library objects that are necessary for performing navigation.

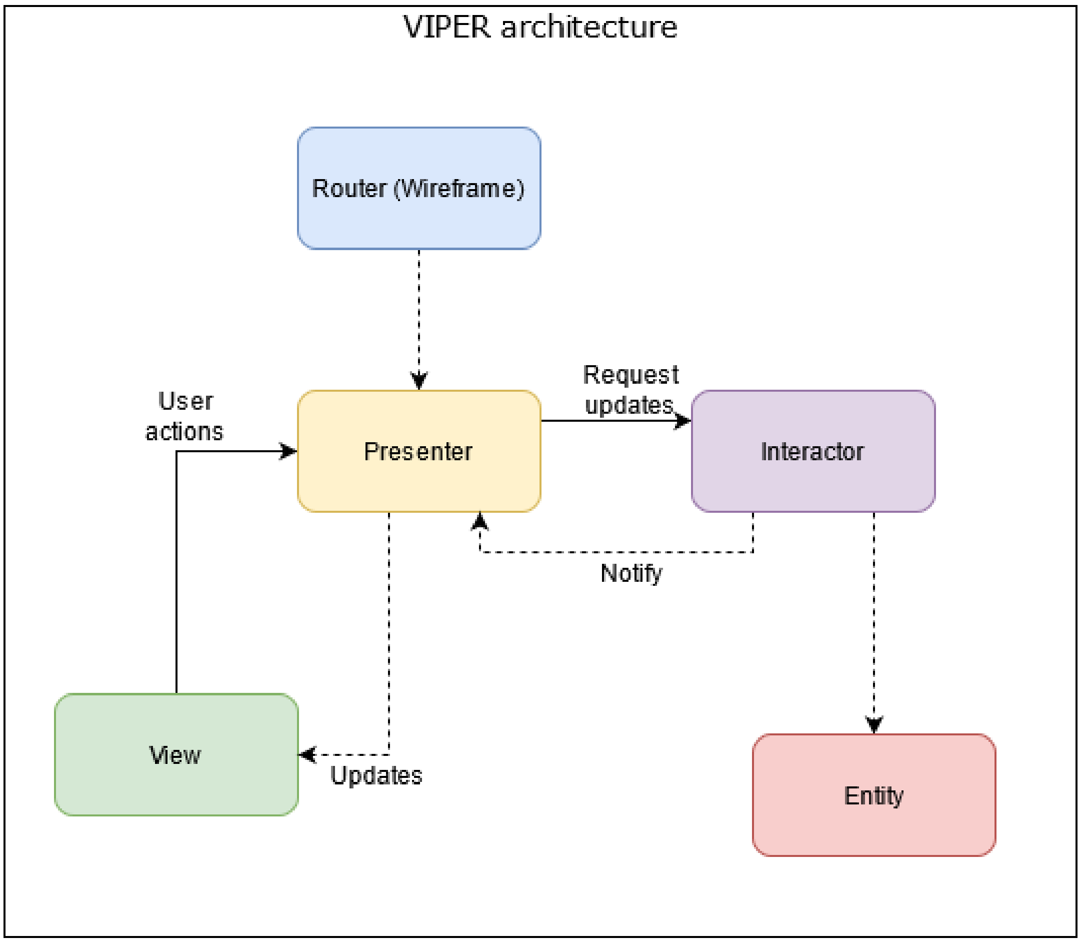

FIGURE 3.1: VIPER architecture.

VIPER architecture gives you flexibility in the way you implement it in your application. For example, you can use data binding between View and Presenter layers to establish a UI presentation layer.

In comparison to MV architecture VIPER requires deep understanding of Clean Architecture principles and why should developers separate different application-level layers. If you know how to effectively use it, VIPER becomes an incredible tool in the hands of experienced developers.

VIPER is a perfect architecture for large enterprise projects with big teams, since it provides an ability to more effectively test the code then MV architectures, independently implement new features and work safely with different business and application logic layers.

The main layers, which stand in architecture's name, perform mostly core logic of the project, but also VIPER includes external interfaces and services, which can be changed during the application development. This external layer includes API services, Workers, DB Managers and other objects, which have no impact on application core logic and features.

3.2 Clean Architecture

Clean architecture is not a concrete pattern, it is an abstract paradigm and set of rules on how to build effective architecture with isolated dependencies and separate applications layers. In this section, we will look on clean architecture from the perspective of Swift and modern iOS development trends.

Usually, Clean architecture is represented as a series of concentric rings and in such abstraction the rules are following: inner layers do not do any application specific-work, they do only pure computations and then using protocols communicate with outer rings (Martin, 2012).

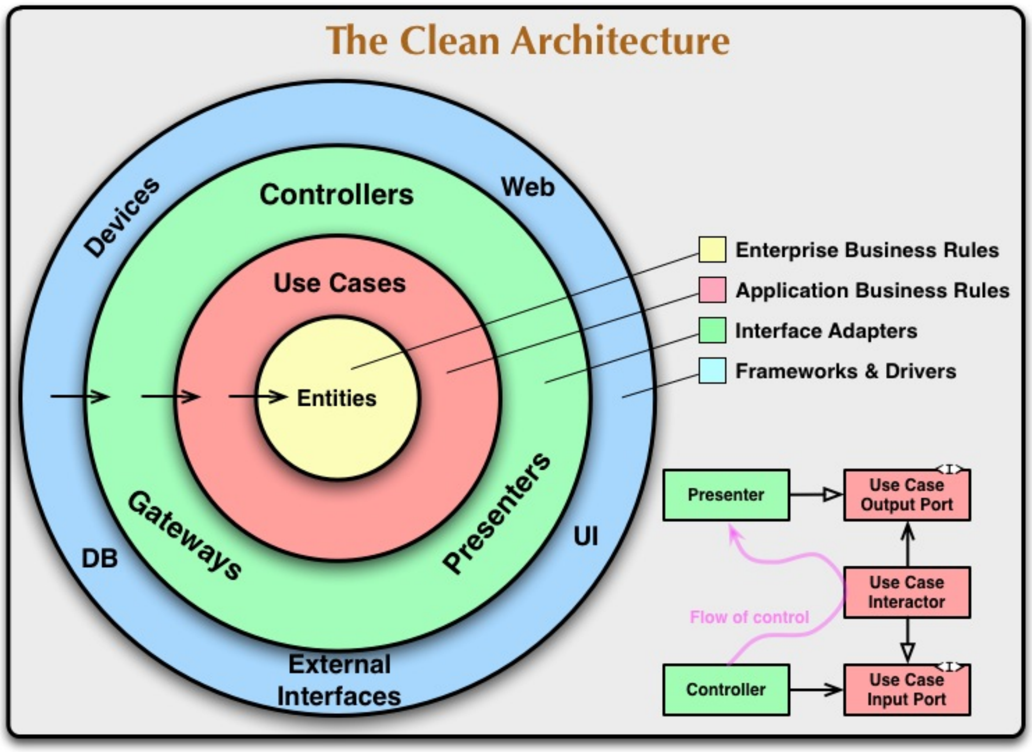

FIGURE 3.2: Clean architecture.

For most iOS applications the Clean architecture layers include following levels (their number may be different, it depends on customisation of clean architecture within specific projects):

1. Entities or Enterprise logic - this layer includes all the internal and core application data models, which are used inside the app. This layer knows nothing about user interaction, navigation, API calls and so on. These objects define core rules and models, for example in Swift these are structs and protocols. If we consider VIPER as Clean architecture, Entity is the layer which perfectly suits these rules.

2. Use cases or Business logic - this level of application layer separation includes business logic of the application. It performs all the application-specific data transformations, computations and manipulates the data models, which are from Enterprise Logic layer. In VIPER the layer which is responsible for business logic is Interactor.

3. Application logic - all the I/O, preparing data for presentation and similar operations are done within this layer. Moreover, navigation tasks also belong to this layer. So, the application logic layer holds View, Presenter and Wireframe from VIPER, since these objects cover all the application logic.

4. The last layer is External interfaces layer - these are API services, DB Managers and other tools, which are simple drivers for the previous three layers. They provide the concrete data, but always can be replaced with another framework or service, so 'outer ring' is the place for these objects.

Much more detailed review on Clean Architecture you can find in the famous among software developers book by Robert C. Martin 'Clean Code'.

As for me, understanding the concept of Clean architecture is essential for each iOS developer, because with this knowledge a developer can build custom architecture, which will perfectly suit his needs in specific applications.

Clean Architectures such as VIPER, Clean Swift or some custom architectures which follow the above principles are the right choice for enterprise projects with a number of features. Such applications often have very serious implementation and separate different layers to the limit. Clean architecture on massive projects also opens the door to create different teams with distinct responsibilities and to maintain the development process avoiding common Model-View problems.

Chapter 4

Conclusion

In this work I tried to analyze and compare most popular architectures for iOS development. Each architecture has its advantages and disadvantages and can nicely suit to build specific projects.

Summing up, if you have to choose which architecture to use in your project, firstly think of the size and scale of your application. If the application is for personal use or need to be done in short terms, MVC or MVP would be the best choice here. Whereas if you plan to create fancy UI components and cool animation or want to try working with a fast-growing and popular approach, then try MVVM. This architecture gives a lot of possibilities to develop applications using new technologies and trying to implement interesting solutions.

Last but not least, VIPER and Clean Architecture can perfectly fit into enterprise projects or large applications with lots of functionality. Their versatility and responsibility segregation can help you build your own custom architecture, which will be clean, testable and perfectly suitable to your purpose.

4.1 Further works

There are lots of other architectures, which were not included in this work. I think lots of them have a great potential, however some of them are not very popular and others are quite specific to compare with Model-View or Clean architecture.

In the near future I plan to analyze such architectures like RIBs (developed by Uber), Lotus (clean modification of MV), Redux (came to us from React Web Development).

Bibliography

Adibowo, Sasmito (2020). *How to Implement VIPER Clean Architecture in an iOS App*. URL: https://cutecoder.org/programming/how-implement-viper-clean-architecture-ios/ (visited on 05/15/2021).

Aljamea, Mariam and Mohammad Alkandari (2018). "MMVMi: A validation model for MVC and MVVM design patterns in iOS applications". In: *IAENG Int. J. Comput. Sci* 45.3, pp. 377–389. (Visited on 05/14/2021).

Dobrean, Dragos and Laura Diosan (2019). "Model View Controller in iOS mobile applications development." In: *SEKE*, pp. 547–716.

HARUN, FIRDAUS BIN (2019). "REVIEW OF IOS ARCHITECTURAL PATTERN FOR TESTABILITY, MODIFIABILITY, AND PERFORMANCE QUALITY". In: *Journal of Theoretical and Applied Information Technology* 97.15. (Visited on 05/13/2021).

Luong Nguyen, Khoi Nguyen (2017). "Application of Protocol-Oriented MVVM Architecture in iOS Development". In: (visited on 05/15/2021).

Martin, Robert C. (2012). *The Clean Architecture*. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (visited on 05/14/2021).