Model View Controller in iOS mobile applications development

Dragoş Dobrean and Laura Dioşan
Faculty of Mathematics and Computer Science, Babes Bolyai University
Cluj Napoca, Romania
{dobrean,lauras}@cs.ubbcluj.ro

***Abstract*—Due to the increased number of mobile applications and their popularity, many software developers have begun to focus on mobile platforms. While this focus has positive effects (e.g. a larger developer community, new open source projects, new tools), it also has a down side. With the migration of developers from different software development areas, where they have used other programming paradigms or architectural approaches, the topic of software architecture on mobile platforms become more trending and hype in the mobile development communities. Even though several new architectural solution were proposed for solving some of the issues which arise from using the classical architectural patterns popularised by the creators of the mobile platforms, we want to emphasise the principles of software architecture in mobile computing, why they have to be respected and how their adoption impacts the development process. Therefore, this paper focuses on showing that the Model View Controller (MVC) — one of the most common classical architectural patterns — can be used successfully for building mobile applications and the problems which might arise are by products of the wrong usage of the pattern rather than pattern issues. We show that by analysing the most common architectural misuses of the MVC pattern in both open-source and private projects and offers solutions to those problems.**

***Index Terms*—Apple's Model View Controller (MVC), Mobile Software Architecture, Architectural Smell, iOS.**

I. Introduction

Mobile applications have become an important part in the life of the modern man. The involved devices have become indispensable companions in our lives. We use them for social interactions as well as for business activities, for increasing our productivity or for self improving and entertainment. According to GSMA Intelligence two thirds of the world are connected by mobile devices [1]. In order to be able to sustain this high rate of adoption and popularity of the mobile applications, many developers migrated from building other types of software to building mobile applications.

This blend of domains has brought many new trends to the mobile platforms (for instance, functional programming which has become used in large mobile projects [2], [3]). However, this union of domains has also had downsides: new or inexperienced developers of those who have migrated from other platforms did not understand fully how all the mobile development concepts work, how are they supposed to be used, and there were not many places in which they could learn how to properly use those. Mobile software architecture has become greatly affected by this phenomena [4]–[6].

It is an established fact that a good software architecture and design could increase the system quality: performance, evolvability, maintainability and reliability [7]. When weak design decisions affect the software properties, the architecture is often subject to various problems (code smells, design smells or architectural smells). Related work regarding software architectural smells can be found in [8] where the authors showcase 11 architectural smells grouped in 4 different categories (Interface-Based Smells, Change-Based Smells, Concern-Based Smells and Dependency-Based Smells). All of issues can also apply to mobile platforms software applications. In [9] and [10] 4 of the architectural smells are analysed in depth and shown on 2 industrial software systems. Other work has been done in [11] where the authors have analysed the architectural erosion of Open-Source Software projects.

Although the drawbacks of architectural erosion have been already recognized [9], [10], [12], [13], the authors of these studies have focused on classical systems, rather than mobile ones: Velasco et al. [14] presented several architectural smells that are relevant to MVC, while Aniche et al. [15] identified six classes in the context of web applications constructed on MVC pattern.

Our initial effort drives to identify the possible MVC problems in the context of mobile applications, to characterise them and to describe how they can be fixed. In this paper, we aim to answer the following research questions:

- RQ1: What are the problems of MVC in the context of mobile applications?
- RQ2: Is there a classification for the MVC architectural problems?
- RQ3: How can the problems be fixed?

To the best of our knowledge, an MVC analysis in the iOS context was not been performed until now. Previous works [14], [15] are focused on web or classic flavours of MVC and on the problems caused only by the violation of constraints which MVC style defines between its components or layers. In the mobile context, the weaknesses of Android's design have been analysed and a passive flavour of MVC was proposed in [16].

The rest of the paper will focus on the iOS platform on its architectural pattern encouraged by Apple, Model View Controller (MVC) [17]. Architectural issues explained in this

DOI reference number: 10.18293/SEKE2019-048

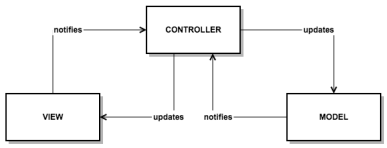

Fig. 1. Apple's Model-View-Controller architectural overview

paper also apply to other platforms which use MVC for building applications (macOS, Windows, etc.), while the naming conventions might be different the concepts and issues are the same. MVC was chosen as is one of the most know presentation architectural pattern while being present on all the mobile applications platforms in various flavours. In addition, MVC is highly versatile, having different flavours as Model-View-View-Model (MVVM) [18], Model View Presenter (MVP) [19], etc.

The following section talks about MVC and Apple's flavour of it. Section

II. Apple's MVC

Model View Controller is one of the most widespread presentational architectural patterns, being used to create desktop, mobile and web applications [20]–[22]. The purpose of MVC is to provide a simple separation of concerns for an application that embeds a user interaction component. One of the most important aspects of this pattern is that the application should work, and fulfil all its requirements even if we remove the View and Controller layers. The data manipulation and the business logic should reside the Model and it should not be affected in any way by those other layers.

Apple's version of MVC [17] is different from the conceptual, generic one [23]. The classic MVC [23] was coined when there did not exist the concept of mobile applications; in order to compensate for this fact, Apple promoted a flavour of MVC, which is better suited for mobile and desktop applications.

It is composed from the same three layers (Model, View, Controllers) and the only thing changed is their way of interaction and the data flow. The layers are more decoupled and it does not rely so heavily on the Observer/Delegate pattern; it is still being used, but it is not as used as intensively as on the classical pattern. In Apple's flavour, the Observer/Delegate pattern is more aimed to provide callbacks from one layer to another than to observe Model layer properties.

The accent in the Apple's flavour of MVC is on the Controller, as can be seen in Fig.

This emphasis shift can also be seen in the way they have named their framework components. At the centre of every iOS application, we find the View Controllers which act as bridges between the data of the application (Model) and the user interfaces (View).

In Fig.

Advantages This flavour of MVC simplifies (from the classic MVC) the data flow between the layers, making the data flow clearer. In addition, it also reduces the coupling between the layers; the link between the Model and The View (from the classic MVC) no longer exits, making the components more isolated.

Disadvantages The Controller layer becomes the central piece of the architecture; it needs to ensure the proper communication between the Model and the View layer and vice-versa. This task makes the Controller layer to grow to be quite complex, which can lead to architectural issues (massive view controllers) as well as OOP issues (violation of single responsibility principle).

III. Analysis

As instances of poor design decision, the architectural smells originate in the improper use of a design solution or of software architecture-level abstractions. In what follows we attempt to facilitate the identification of such problems in the context of mobile applications involving the MVC pattern. We provide a short description of each problem and its causes (trying to answer RQ1), the architectural smell's class it belongs to (as answer to RQ2) and one or more possible solutions (answer to RQ3).

In order to be able to answer those questions we needed to inspect different sized codebases, we have chosen 5 codebases, with different variations of MVC that added architectectural layers from: MVVM [18], MVP [19] VIPER [24].

- Wikipedia – education and information app [25]
- Firefox – a mobile web-browser [26]
- Trust – cryptocurrency wallet [27]
- E-Commerce — private
- Game – private

TABLE I
Codebases size

Application	Blank	Comment	Code	Source
Firefox	23392	18648	100111	open-source
Wikipedia	6933	1473	35640	open-source
Trust	4772	3809	23919	open-source
E-Commerce	7861	3169	20525	private
Game	839	331	2113	private

As can be seen in Table

A. Complexity

1) Issues: Mobile applications have grown to be complex software systems where they do much more than just fetching some data from a web service and displaying them on the screen. While for a simple application the MVC pattern would be sufficient, if we add extra complexity, such as working with databases, caching, virtual reality, audio, photo or video manipulation, we might ran into some issues.

Although using another presentational pattern (e.g. MVVM [18] or MVP [19]) might solve some of the problems, if the application becomes more complex, even these patterns will not be sufficient to maintain it flexible to change, testable and easy to be understood and worked on.

2) Solutions: MVC is an architecture to be used on small-medium sized applications. If we talk about a complex application then an architectural approach needs to be designed in order to fulfil its use-case. MVC provides the basis for this new architecture and its three degree of separation should definitely be implemented. But, in addition and in order to make the codebase maintainable, we might introduce additional layers, such as Presenters from MVP, View - Models from MVVM or Routing objects from VIPER [19], [24], [28], [29].

3) Findings: Trust, E-Commerce and the Game had the clearest defined architecture. Analysing the codebase is easy to get a grasp on how the app works and the codebases were consistent in both naming and design pattern used. Firefox, being the largest codebase and given its functionality, is a more complex one and it is the hardest from the codebases to understand and uses multiple layers. Wikipedia relies on multiple open source libraries and internal UI libraries which introduce extra obfuscation and makes the codebase harder to comprehend.

B. Misunderstandings

1) Issues: Another common problem we encounter when talking about MVC is that people usually have different concepts of what MVC is and how should it be used. People coming from ASP.net MVC [30] development might have a total different idea of how the MVC components should communicate from those who are developing mobile applications. As we have already seen, different companies have different definitions for what MVC is and how it should be used with their frameworks. This problem has a real impact when developers migrate from one platform to another and try to use the same MVC definitions on a new framework.

Almost all people with some knowledge about the MVC agree that the Model should contain data useful for application [31], the View is responsible with presenting the data to the user and the Controller layer acts as a mediator of some sort between the other two layers. Those are very vague definitions of MVC and 2 people can disagree on what should be put in the Model and what should reside in the View or Controller layer.

Developers of MVC frameworks usually give their definition of what MVC meant for them or how should this be used; however, those definitions usually are vague as well. The reason of this ambiguity can be rooted in the generality of the frameworks, which should adapt to many types of applications. A constrained architecture, which would fulfil all potential use-cases is not feasible and it is considered redundant for applications that do not need a high level of architectural complexity.

2) Solutions: The first step into ensuring that an architectural pattern is correctly implemented is to have clear definitions of its elements and everyone involved in the project to be well aware of. In order for this to happen, it is important that the lead developer or the system architect to understand clearly the scope of the product and to be able to draw architectural guidelines which would fit the project.

3) Findings: From the analysed projects, Trust, Firefox and the E-Commerce app were the ones in which there was a clear defined architecture which could be inferred from the codebase and it was consistently used. The Game app was the smallest and its architecture didn't require any specific guidelines given its complexity. In the case of Wikipedia, the code was not consistent, and the guidelines were not clear.

C. Model

1) Issues: Most of the problems which appear in this layer are design pattern issues; the usage of too many singleton objects and the violation of SOLID principles [32] are the root cause of the problems which can appear at this level. The result will be a damaged architecture at a micro level — high coupling between items in the same layer.

Among the common mistakes in the model are the fact that objects which interact with a database or a web backend service have reference to the ones using those (usually View Controllers). These references can create retain cycles — Dependency-Based Smells [8] — and also impact the MVC architecture by making the Model layer have knowledge about the Controller layer.

The problems which appear at this level are usually from the Interface-Based Smells category as defined in [8], [11]. Is not uncommon to find Ambiguous Interfaces or Concern Overload where a component performs a large amount of tasks and have a scarce number of interfaces. For instance, the objects which communicate with the backend for fetching data are most of the time responsible for creating the connection, converting the input parameters to what types of information does the backend service expect, parsing of the response it receives and mapping it to a codebase defined entity.

2) Solutions: This issue can be solved using the Observer/Delegate pattern, where the Model layer provides callbacks for its events and the Controller layer takes various actions based on those events.

3) Findings: All the analysed codebases presented issue on the Model layer as each one of them has wrong, direct dependencies between the Model layer and the View or Controller layer. The most problematic ones were the E-commerce and Firefox. In all the codebases excepting the Game, we find the Concern Overload and the Ambiguous Interfaces smells [8], [11].

D. View

1) Issues: In large projects, which do not have major architectural issues, the Controller objects configure the Views by directly passing the Model item as an argument. This common practice creates a dependency between the View and the Model, which is not presented in the Apple's way of defining the MVC (Fig.

An example would be a list of new movies in a booking application: the cell that is responsible for displaying a new movie will usually receive from the Model, a Movie entity, which contains much more data then what is needed to be displayed (the ID from the database, a list of actors, number

of people who already booked it, etc.). This is an overlooked issue with MVC and usually the developers accept it, even if this is an architectural mistake nevertheless.

The mentioned problem belongs to Co-change Coupling smells [11], an architectural issue which occurs frequently at this level. The coupling is predominantly done between the View and the Model layer. However, this can also appear in the View and Controller layer.

2) Solutions: In order to overcome this difficulty, new objects can be defined for keeping the configuration of the view (when the view needs a lot of configuration information from the model), or this information can be passed as parameters to the view using primitive types.

The new defined items for the configuration of the View rise another problem: where should those items reside? They know nothing about the View so is not in the View layer; however, they are only used and have meaning in a context in which those Views exist. An approach to solve this problem would be to treat these items as belonging to the Model as they handle the business logic display part, they can be seen as mappers between entities and views or data transfer objects.

Another approach, if the developed application needs this kind of complexity, is to use another architectural pattern namely MVP [31], which inherits from MVC; basically, it is a variation of MVC, where there is a new layer for configuration objects, called Presenter. The Presenter, however, is a more specialised object: besides configuration information, it also contains information regarding the state of the View (selected, unselected, highlighted, whether or not some of the fields should be pre-filled etc.).

3) Findings: All the analysed codebases have shown Co-change Coupling smells, the most severe ones was the E-commerce one. The open-source apps also exhibited this issue, however at a much lower degree.

E. Coordinating Controllers

1) Issues: Just like in the case of Apple's MVC, there are different flavours of MVC where there is a combination of roles (View and Controller) into a single entity called View Controller. This entity owns the View and it responds to its events. The View Controller is responsible for responding to input received from the View and for displaying and moving those Views on the screen. Those kind of controlling objects usually derive from a superclass. For instance, on the iOS SDK, the superclass is UIViewController, on Android we have the Activity superclass.

There are cases where the complexity of the application requires another kind of controller objects — Coordinating Controllers. Coordinating type of controllers are simple objects that manage the application; they usually decide when a certain action should happen and keep track of the state of the application [33]. Those kinds of objects are responsible for deciding on what state (flow) of the application to go next (when a certain event occurred) based on the current state, for setting up the initial state and managing the lifecycle of contained objects.

By flow and state we mean what use case scenario is presented on the screen at a certain moment in time; flow is a broad term and in the context of this paper we are using it to describe a use case (e.g. sign up), if we were to have a higher granularity, the flow can be split in multiple sub-flows (e.g. the forgot password of the sign up flow).

Therefore, the Controller layer can be split into two categories: View Controllers and Coordinating Controllers. The View Controller objects have come to be generally accepted as the Controller objects by most of the practitioners in this field. However, this is not always the case and there is an important distinction between Coordinating Controller objects and View Controllers [17].

Unfortunately, this degree of separation in the context of Controller layer is not so well understood on the iOS platform. All these concepts appear in AppKit development scene (desktop application for macOS), as this platform is older and more evolved. Usually this form of separation within the Controller layer is not needed, as the applications are not complex enough to justify it. The problems start arising when the application becomes complex and the people working have a lack of architectural knowledge on how to scale it or the architectural state in which they need to arrive is unknown or insufficiently defined.

Frequently, the responsibilities of Coordinating Controllers get stuffed in the View Controller objects increasing their complexity and changing their purpose as now, they also have to take care of knowing the state of the application and correctly transitioning between the states in every possible configuration. By taking this responsibility in other custom objects (Coordinating Controllers), the View Controller object become slimmer and they are no longer depending on each-other.

This practice is fairly popular or familiar and is usually implemented in applications where a clear architectural guideline is not defined or not sufficiently described and defined for all the potential corner cases. A lighter common version of this coordination is to have an object which all the View Controllers inherit from, where all the common navigation flows are stacked in.

This sort of behaviour (merging responsibilities) is common for small applications where the UI is quite simple (one–three screens), where the extra Coordinating Controller objects would not provide real value, or for beginner developers. Most of the applications which are fairly complex have multiple flows (sign up, sing in, browse items, add to cart, checkout, previous orders, feedback, settings, profile, etc.). These applications are the ones which suffer massively from the lack of coordinating layer as their View Controller objects become bloated with navigation and configuration logic. This kind of complexity creates architectural issues especially when the application needs to be changed because many components fulfil the same functionality, for instance the correct navigation from one screen to another (Scattered parasitic functionality [8]).

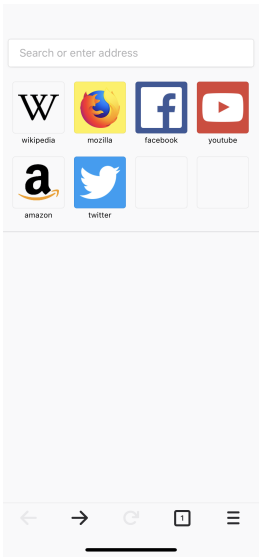

Fig. 2. Firefox iOS application screenshot [26]

2) Solution: The solution to this problem (complex application with multiple use-cases and flows) is to have multiple Coordinating Controller objects for every flow of the application or for every sub-flow of the application; each of these flows will have a single, well defined use case (login in the user, uploading a picture, making a payment, etc.). All the application's flows and navigation will be then expressed via those building blocks (Coordinating Controller objects for certain flows or sub-flows). By using this approach, we reduce the complexity of View Controller objects. They become concerned only with displaying the data and mediating between View and Model layers. All the navigation and configuration logic now resides in the Coordinating Controller objects. In addition, we can easily change the flows of the application even at runtime, we can Unit Test the navigation from one screen to another and the correct configuration of the View Controller objects, which, in the case of massive view controllers, is rather hard.

3) Findings: Firefox, Trust and the E-commerce apps were the ones in which the Coordinating Controllers were correctly used. Wikipedia was the worst analysed app from the Coordinating controllers point of view, the Scattered parasitic functionality [8] is predominantly present in the codebase.

F. View Controllers

1) Issues: Another issue which is overlooked is that developers usually use one view controller per screen (the UI elements shown on the full size of the screen). While this is the right approach for simple screens such as a "Terms and conditions" screens or even a "Login" screen, if we talk about complex UI interfaces (e.g. the browse screen from Firefox) this is totally wrong.

The browse screen from the Firefox application (see Fig.

A large amount of the problems which we encounter in the MVC approach on iOS deals with the View Controller. In fact, it has access to both View and Model layers and acts like a binder between them; an example would be if the data obtained from the Model is not well formatted, the View Controller will format it for the View and this is not clearly its responsibility. A View Controller should only be concerned with presentational aspects of a certain part of the application and for handling the user input received from the View. Obviously problems can appear at other levels, as well on Coordinating Controllers or Model level, but these usually, like in the case of View Controllers, have the root cause the low granularity of the architectural components and can be solved by increasing the granularity of the elements (splitting a certain item in multiple others and use the Composition design pattern).

In addition, the View Controller objects are also bloated with handling View logic and states. By view logic and state we mean keeping the internal state of the view which cannot be inherited from the Model. For instance, knowing which items were selected on the screen, what slider is enabled etc., before applying these changes to the Model. This sort of logic should be implemented in custom objects; most of the times the MVP pattern is used for solving this issue, but as a workaround in MVC, these can reside in subclasses of View components or in custom objects defined in the Model layer.

At this level, the major architectural smell is Concern Overloading [11], as like previously shown, the View Controller object become bloated with an excessive amount of responsibilities.

2) Solutions: The solution to this problem is to use multiple view controllers for the UI elements; for instance, we could have a View Controller object responsible for the turn by turn navigation advices, we could have another one for the map and so on. Furthermore, if these elements are complex by their own, they could be split further in more View Controller objects which should respect the single responsibility principle. By using this approach we would obtain view controllers that respect the single principle responsibility ensures a good separation of concerns, they contain less code, and they become testable.

As in the case of Coordinating Controllers where we could have Coordinating Controller objects, which depend on other Coordinating Controller objects we can apply the same logic to creating user interfaces and using multiple child View Controller objects to construct a single screen of the application. By using this approach, each View Controller object will have single responsibility and purpose.

3) Findings: All the codebases shown signs of Concern Overloading [11], the issues are however bigger as the codebase increased. In the case of the apps which were using Coordinating Controllers (E-Commerce, Trust, Firefox) the issues were lower than in the case of Wikipedia where the View Controller classes were way more complex as they also had to handle navigation logic.

IV. Conclusions

By our study we have tried to provide interesting insights about several common problems of MVC for both mobile developers and scientific community which are commonly found in open-source or private projects. We describe these problems in detail as well as their corresponding architectural smells. Furthermore, several solutions to those problems have been proposed which shed some light on architectural corner cases which were less explored by practitioners.

As we have shown previously in this paper, MVC can be used as the presentational software architecture for a mobile application. If the concepts are implemented correctly this does not produce any of the popular issues, neither massive view controllers nor the violation of the single responsibility principle.

What is important to be understood is that based on the complexity of the application the entities in the MVC architecture should be more granular, in order to be flexible, testable and maintainable. Based on this complexity, new types of layers or sublayers can appear which are close related to the requirements of the application.

Based on the observations made throughout many years of developing commercially those kinds of applications, the presentational architectural concept used was never an issue for the flexibility, extensibility and testability of the application; the issue always came from its bad implementation, or the misusage of programming language featurese

Our further work will continue on developing tools for ensuring that a certain architectural pattern or certain architectural rules are respected with every commit made by a developer. By following this direction we can educate developers regarding the architectural aspects of a mobile software application, we will help them produce cheaper and cleaner code.

References

[1] GSMA. (2017) Global mobile trends. link.

[2] A. Cowkur. (2017) Functional programming for Android developers. link.

[3] ObjC.io. (2016) Functional programming. link.

[4] E. Bessarabova. (2017) MVP vs MVC vs MVVM vs VIPER. What is better for iOS development? link.

[5] K. Kocsis. (2018) Architectural patterns, MVC, MVVM: What is the hype all about? link.

[6] E. Maxwell. (2017) MVC vs. MVP vs. MVVM on Android. link.

[7] H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, and N. Medvidovic, "Software architectural principles in contemporary mobile software: from conception to practice," *Journal of Systems and Software*, vol. 119, pp. 31–44, 2016.

[8] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic, "Relating architectural decay and sustainability of software systems," in *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*. IEEE, 2016, pp. 178–181.

[9] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE, 2009, pp. 255–258.

[10] ——, "Toward a catalogue of architectural bad smells," in *International Conference on the Quality of Software Architectures*. Springer, 2009, pp. 146–162.

[11] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 176–17 609.

[12] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic, "Mapping architectural decay instances to dependency models," in *Proceedings of the 4th International Workshop on Managing Technical Debt*. IEEE Press, 2013, pp. 39–46.

[13] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software engineering notes*, vol. 17, no. 4, pp. 40–52, 1992.

[14] P. Velasco-Elizondo, L. Castañeda-Calvillo, A. García-Fernandez, and S. Vazquez-Reyes, "Towards detecting MVC architectural smells," in *International Conference on Software Process Improvement*. Springer, 2017, pp. 251–260.

[15] M. Aniche, G. Bavota, C. Treude, A. Van Deursen, and M. A. Gerosa, "A validated set of smells in Model-View-Controller architectures," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 233–243.

[16] K. Sokolova, M. Lemercier, and L. Garcia, "Towards high quality mobile applications: Android passive MVC architecture," *International Journal On Advances in Software*, vol. 7, no. 2, pp. 123–138, 2014.

[17] Apple. (2012) Model-View-Controller. link.

[18] A. Sinhal. (2017) MVC, MVP and MVVM design pattern. link.

[19] M. Potel, "MVP: Model-View-Presenter the taligent programming model for C++ and Java," *Taligent Inc*, p. 20, 1996.

[20] R. Eckstein. (2013) Java SE application design with MVC. link.

[21] D. Plakalovic and D. Simic, "Applying MVC and PAC patterns in mobile applications," *arXiv preprint arXiv:1001.3489*, 2010.

[22] M. J. Yuan, *Enterprise J2ME: developing mobile Java applications*. Prentice Hall Professional, 2004.

[23] G. E. Krasner, S. T. Pope *et al.*, "A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[24] S. M. Alam. (2017) VIPER design pattern for iOS application development. link.

[25] Wikimedia. (2018) Wikipedia iOS application. link.

[26] Mozilla. (2018) Firefox iOS application. link.

[27] Trust. (2018) Trust wallet iOS application. link.

[28] R. Garofalo, *Building enterprise applications with Windows Presentation Foundation and the Model View View Model Pattern*. Microsoft Press, 2011.

[29] ObjC.io. (2014) Architecting iOS apps with VIPER. link.

[30] Microsoft. (2013) ASP.NET MVC overview. link.

[31] M. Fowler. (2006) GUI architectures. link.

[32] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[33] Apple. (2012) Controller. link.