STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIV**, Number 2, 2019
DOI: 10.24193/subbi.2019.2.04

A COMPARATIVE STUDY OF SOFTWARE ARCHITECTURES IN MOBILE APPLICATIONS

DRAGOŞ DOBREAN AND LAURA DIOŞAN

Abstract. The mobile market grows larger year by year and at the core of those devices, we have the mobile applications that push the technological advancement forward continuously. Due to the increased hardware performance and the popularity of those devices as well as advancements in their operating systems, mobile applications have grown to be complex projects with many dependencies and large teams working on them. As the application becomes bigger and more complex, the problem of choosing the right software architecture arises. This study focuses on an analysis of the most commonly used architectural patterns on mobile applications highlighting their features and flaws. Moreover, it also presents a comparison between them when implementing a medium-sized application. The usage of the appropriate architecture can simplify the work of developers and enable the creation of sustainable applications and the improvement of the software?s capacity to endure and evolve over time.

1. Introduction

The market of operating systems for mobile devices is shared between Android (Google) and iOS (Apple), together they cover over 95% of it [23]. Since they cover so much of the market, those operating systems have to run on both high end devices and low end ones. Mobile applications nowadays have short release cycles, they pack the latest technologies and trends for the high end devices and they also maintain the support for the old or low end devices.

In order to be able to design a software system that can work well and can be implemented efficiently under the given circumstances, there is a need for an architectural solution that matches the scope of the desired system.

There are some architectural patterns pushed by the creators of those platforms; Apple promotes MVC in their iOS framework while Google promotes

Received by the editors: October 22, 2019.

2010 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.11 [**Software engineering**]: Software architectures – *Domain-specific arhitectures*.

Key words and phrases. mobile applications, iOS, software implementation, software architecture, programming techniques.

MVP on Android, but they do not scale well and are not suited for all classes of applications. Moreover, most of the time they are wrongly used resulting in massive classes that heavily violate the single responsibility principle, have low cohesion and high coupling. Those kinds of wrongly designed applications are really hard to test and while their codebase increases, the development time needed for adding new features increases drastically as well. Furthermore, the whole development process becomes laborious as the codebase becomes hard to understand. It usually lacks testing and it is very difficult to implement a new feature or modifying one without breaking another part of the application.

This article will focus on analysing the most common software architectures used on the iOS platform by taking into account criteria like reusability, flexibility, testability, dependency among components, development costs. The strong and weak features of each architecture will be discussed, while answering three important questions, which lay the foundation for this study:

- Q1: Why is the software architecture important in mobile applications?
- Q2: What does good software architecture mean in the context of mobile applications?
- Q3: How can a mobile platform software architecture be analysed and benchmarked?

Section 2 talks about the most commonly used mobile architectures on iOS platform highlighting their strong and weak points; section 3 presents our process of analysing those architectures, while section 4 showcases our findings in regard to the questions above enunciated.

2. Details of patterns & comparative analysis

In [12], one of the first papers that described and compared two presentation patterns for designing mobile application — MVC and Presentation-Abstraction-Control (PAC), the authors have emphasised the conditions facing mobile application and they have concluded that the selection of a particular software pattern for the user interface architecture depends on the class of mobile application.

Another MVC-based architecture, called balanced MVC architecture, has been proposed in [7] for service-based mobile applications. The proposed architecture is aimed to divide the kernel application optimally between the client and the server. Again, the authors have remarked the specificity of the proposed architecture for different types of applications, but no other design patterns have been taken into account and analysed.

A unified architecture model adapted to Android development called Extended MVC has been proposed in [19]. The adaptation regards the specificity of the mobile applications. The authors have tested their approach by considering ten devices of various physical specifications (versions of Android system, screen resolutions, internal storage, CPU, RAM, etc.) and they have concluded that their pattern improves the flexibility of the mobile application (without using some metrics for evaluating it).

In [21] and [22] the authors have surveyed several widely used architectural design patterns (MVC, PAC, HMVC [1], MVP, MVVM) involved in the development of mobile applications. Furthermore, the authors have proposed an MVC-based design pattern particularly adapted to the Android system (called Android Passive MVC) and they have evaluated its quality in terms of maintainability, extensibility and reusability with scenario-based software architecture evaluation method. The authors have remarked (and somehow quantified) the reduced complexity of the mobile application developed by integrating the proposed architecture.

VIPER is another alternative architecture proposed by MutualMobile in [10] and comes as an alternative to Clean Architecture from Android on iOS. In [16] the authors highlight the importance of using specialised architectures rather than the "default" ones and analyse the re-architecting process of Coursera's mobile application, where they have chosen VIPER as their architectural solution.

Other technical surveys about architectural patterns can be identified by taking into account the industrial/technical blogs [24], [20].

2.1. **MVC.** The Model View Controller is one of the most versatile and used software architectural patterns. It has been firstly used in Smalltalk and was later adopted by Objective-C and other programming languages such as Java and Ruby [15], [14], [2]. It is used for developing desktop, web and mobile applications [17].

This pattern is the one that is promoted by Apple with its iOS platform, encouraging the development of the applications that pursue it. Many frameworks available from Apple for development purposes follow this pattern and, when using them, custom objects are required to play an MVC role. However, the simple usage of these frameworks do not guarantee that all MVC principles are respected. Many frameworks available from Apple for development purposes follow this pattern [8] and when using them custom objects are required to play an MVC role.

Apple's MVC is a little bit different from the classic MVC as shown in Figure 1. In the classic MVC implementation, the model communicates with

[1]Hierarchical Model View Controller

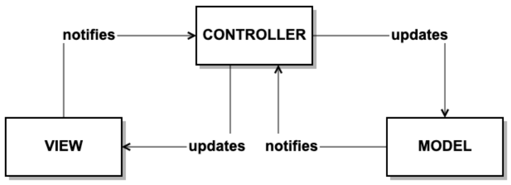

Figure 1. Model-View-Controller architectural overview

the view, while in this flavour of MVC the controller acts as a mediator between the model and the view, is responsible for updating the model as well as the view and reacting to notifications from both the model and the view.

The controller has a more active role than in the classic pattern being the bridge between the view and the model. Because this type of object is concerned with how and when to display certain data on the screen and how to react to user interaction it has been named "ViewController". In addition to this, the data and event flow in this flavour of MVC is linear, while in the classic architecture the flow is circular.

2.2. **MVP.** The Model-View-Presenter architecture has been long used in other software development areas not only on mobile platforms. The principles behind this pattern were not designed from scratch and it came as flavour of MVC bringing in some advancements. This pattern can also be adapted to a large set of applications such as client/server or multi-tier applications [13].

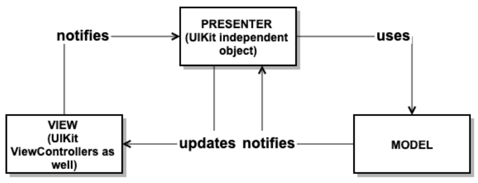
PRESENTER (UIKit independent object)
VIEW (UIKit ViewControllers as well)
MODEL
notifies
uses
updates
notifies

Figure 2. Model-View-Presenter architectural overview

The whole pattern is built with the idea that the actions in the application should be driven by the user interaction, by the view layer rather than by the controller. It is composed of three major types of components: the model that handles all the data, the view that takes care of the interaction with the user, the presenter that is responsible for connecting elements, as can be seen in Figure 2.

MVC and MVP look very much alike and the differences are subtle, that is why MVP comes as a flavour of MVC and not as a new concept, both of them being presentational patterns. While they might look alike, there are differences and advantages in using one or another.

In MVP the presenter is responsible for manipulating the views and they communicate through interfaces, the views being decoupled from the presenters and vice versa. In the world of MVC, all the communication between the views and models is done through controllers, the elements are more tightly coupled. The controller receives an event from the view layer, it does some processing, it might manipulate the models and updates the views accordingly. Another difference is the fact that in MVC the views are dumb objects, they do not contain processing code as contrary to what happens in the MVP pattern where the views have to communicate with the presenter.

2.3. **MVVM.** Model-View-View-Model is another architectural pattern from the MV family, which is heavily used on mobile applications. This pattern has gained a lot of attention and has been implemented in many applications because it addresses the problem of massive view controllers [6] and, it also works well with reactive programming [4].

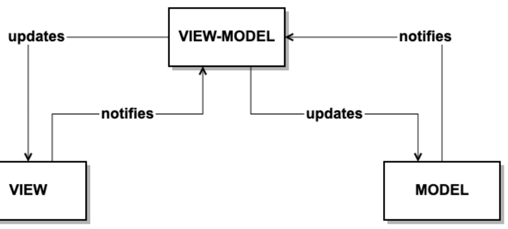

FIGURE 3. Model-View-View-Model architectural overview

MVVM tries to solve some of the problems that might lead to the massive view controllers by using a new layer between the Model and the Controller called View Model as shown in Figure 3. The purpose of this View Model is to take a model object and apply all the transformations and presentation logic to its attributes such that those can be easily presented by the view, for instance, transforming a date into a formatted string. By using this approach the controllers become less bloated with UI configurations and mappings, becoming lighter.

This architectural pattern works very well with reactive programming because the idea behind this architectural concept is that every change done to the model should be automatically reflected in the View through the View Model. The task of propagating this information is easily achieved with the use of reactive programming or by language features such as Key-Value-Observing [1]. MVVM is also compatible with the MVC as it just adds an extra layer that is responsible for configuring the View by mapping various values and applying some business logic on the Model.

2.4. **VIPER.** VIPER stands for View Interactor Presenter Entity Routing and it is a software architecture used in large mobile applications. VIPER does not come from MV family [10], [16], [5], [18]. As shown in Figure 4 it uses five layers of abstraction for separating concerns in the application. It does that for solving problems that come with using a classical MVC architecture where there is no clear layer where the business logic should be placed. VIPER respects the principles of a Clean Architecture [9] and it can be considered a pattern for the whole application (not only a presenter pattern, like the previously described ones).

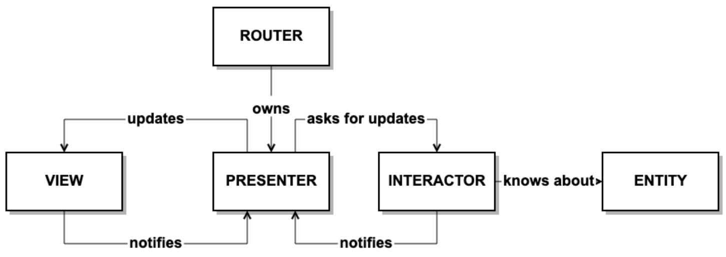

FIGURE 4. VIPER architectural overview

The software systems built using this architecture resemble a game of LEGO. A complete application is built from multiple VIPER modules, the size of those modules depending on the granularity sought. Each component has a well defined and single concern, this architecture is built on the Single Responsibility Principle. The view is only responsible for displaying the items it receives from the presenter.

The presenter works closely with the interactor and prepares the content it receives from the interactor for the view so that this component can display it. The presenter is also responsible for reacting to events from the view and requesting new data from the interactor.

The business logic is contained in the interactor; its responsibility is to manipulate the entity objects. All the logic should be independent on any UI components and all its behaviour should be portable to other platforms.

The entity layer contains the items with which the business logic works and it is related to the model in the MVC. The navigation from one view to another is shared between the presenter and an object which handles the navigation

	MVC	MVP	MVVM	VIPER
Model	Handles data & encodes the model objects			
	Is unaware of anything except itself			
	Handles business logic			Doesn't handle business logic
	Is manipulated by Controller	Is manipulated by Presenter	Is manipulated by ViewModel	Is manipulated by Interactor
View	Presents the interface			
	Handles user input actions			
	Is not Model aware			
	Is passive	Is active	Is passive	Is passive
	Does not have any knowledge about Controller	Has a reference of the Presenter	Has a reference of the ViewModel	Has a reference of the Presenter
Controller / Presenter / ViewModel / Presenter	Handles application navigation			Does not directly handle application navigation
	Contains logic to react to user inputs			
	Updates the views			
	Handles application logic			Does not handle application logic
	Mediator (Model-View)			Mediator (Interactor-View)
	Directly interacts with the Model			Does not interact directly with Model (goes trough Interactor)
	Handles business logic for the view		Contains some business logic (controller logic)	
	Is aware of the View		Knows nothing about the view	Is aware of the View
	Calls methods on the View to notify it to update itself	Calls methods on the View to notify it to update itself	Exposes change events when the state changes	Calls methods on the View to notify it to update itself
	Draws the view trough direct interaction	Drives the view with an interface	Draws the view trough some data binding	Draws the view trough direct interaction
	Tightly coupled with the view	Is decoupled from the view	Not tied to a specific view	Tightly coupled with the view
Interactor				Handles business logic
Router				Handles navigation logic

Table 1. Findings after analysis

stack. The presenter is responsible for deciding when to navigate to another view while the routing object is creating the actual transition.

2.5. **Findings after analysis.** We have already seen that the differences among these patterns are relatively small, but they are significant. To better emphasize the most important characteristics of these patterns, we resume them in Table 1.

3. Analysis and benchmark

After the retrospective of the most used software architectures on the iOS platform, we have implemented them in a medium sized iOS application. The application has eight different screens all of which have custom UI components such as lists, buttons, animations, views as shown in Figure 5.

The purpose of the application is to highlight the codebase complexity and the potential problems which might arise from the usage of those architectures in a mobile application. The application is a simple game and one of its most important functional requirements is the ability to send a messages between players, while maintaining a stopwatch which should be synchronised with the

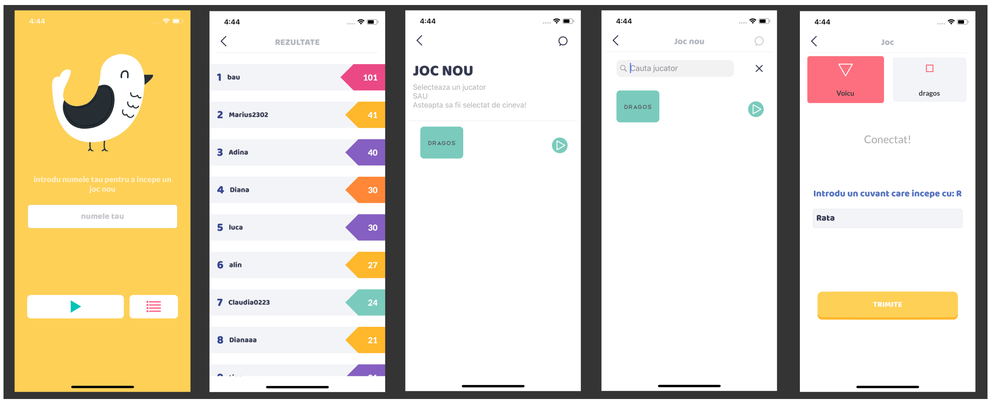

FIGURE 5. Implemented iOS application

one from the peer player. The actual functionality of the application is less important, its choice being made in order to better exemplify the analysed concepts.

After analysing the top 50 free applications from the iOS App Store on various categories, this type of application was chosen as representative for the following reasons:

- it heavily relies on network operations;
- it has a user interface which needs to be adapted constantly and dynamically based on the events its receives from the peer;
- it uses open source libraries which have different architectures and, sometimes, even a different programming language than the ones used in the development of the application.

After implementing the application with all the software architectures described in section 2 we have observed that the MVC is one of the most common architectures as it is easy to follow the pattern imposed by the iOS framework and develop the applications based on the blueprint they provide, as shown in Table 2 (column 2).

However, this can lead to the problem of massive view controllers, as usually developers are not using multiple view controllers for the same user interface page (screen). The main cause of this misusage is that there are not many resources in the literature in which this idea is promoted and most of the beginner developers are unaware of this feature.

In the implemented application we have respected the MVC pattern as described and while we did not produce any massive view controller classes, it was clear for us why the problem might occur. The requirements for the implemented application were clear from the beginning and we have not started to implement extra features on an old codebase. The introduction of new features and changing the development team can lead to the above stated problem, because it is easier to add a something on top of an fully working system than to refactor it and do things properly, in concordance with its architecture. This is especially true for small features such as adding an extra button or a new label.

The MVP offers another flavour of MVC and, while the problem of massive view controllers is somehow resolved, it is easy to create massive presenters classes just like in the case of MVC. While MVP-based approach is a little better than creating massive view controllers, because those presenters are plain objects and usually do not inherit from a superclass or receive callbacks regarding the user interaction actions as view controllers do, this makes them more testable. However, without proper separation and design, we could easily end up with massive presenters, which take the responsibility of a massive view controller, where we also require extra code for passing the user interaction events from the view to the presenter, which is another task that could introduce bugs and increase the development time.

MVVM comes and adds another layer of abstraction to the classic MVC architecture. It binds the models to the views and vice-versa by using another layer of abstraction, the view-model. This approach takes some of the complexity away from the view controllers. We found out that it felt like we are over-engineering when using this approach for small view components, which have only a label or some minimal information. This approach surely makes sense for components that are complex and they need to display easily large amounts of information or for the views with multiple states; however, for light components the amount of work necessary for implementing it does not always justify its advantage.

Another important aspect of MVVM regards the usage of a third party library (Table 2, column 3), which is usually required for implementing the Observer pattern in complex applications. In the case of iOS, when developing an application in Objective-C, this mechanism was already built in the language, however on Swift we can no longer use this approach without heavily relaying on the Objective-C runtime. That is why most of the applications which use this pattern relay on a third party library for implementing the synchronised behaviour and, thus, adding extra complexity to the overall project.

In addition to this, we can see that the MVVM pattern is the only one that implements the synchronicity between the Model and the View layer (Table 2, column 5) which means that when a change occurs to the Model layer it will be automatically forwarded to the View layer and that usually results in an alternated UI as well.

ARCHITECTURE	Easy of use / follow ↑	Requires extra libraries	No. of layers	Synchronicity between MV layers	Complexity no. of lines ↓	Development costs ↓	Granularity	Strong feature	Weak feature
MVC	*****	NO	3	Async	Base	**	Coarse-grained	simple concept	massive view controllers
MVP	****	NO	3	Async	Base * 1.078	***	Coarse-grained	extra flexibility over MVC	massive classes increased complexity
MVVM	****	YES - usualy	4	Sync	Base * 1.065	****	Coarse-grained	increased testability	complex external dependencies
VIPER	**	NO	5	Async	Base * 1.185	*****	Fine-grained	increased granularity and testability	very complex

TABLE 2. iOS Software architectures comparision (↑ - maximum criterion, ↓ - minimum criterion)

The pattern with the most granularity the ones we have analysed is VIPER. The separation of concerns done in this approach tries to ensure that the architecture of the application will erode at a slower pace (see column 8 of Table 2). It provides the greatest flexibility of all the presented architectures and it also the easier and most testable one. Nonetheless, the flexibility and granularity come at the cost of more code written, more layers and the most complex concepts. We also found out that for applications that have a short lifecycle this approach can be costly from a development perspective, being the most time consuming and the implementation which required the most skilful developers.

In regards to development costs Table 2 (column 7) shows that more commonly used architectures such as MVC or MVP have a smaller development cost than the ones which are more complex such as VIPER or MVVM. While ranking the architectures, we have looked at the following aspects: the architectural skills required by the developers to work on a codebase that implements a certain architecture, the ease of implementing new features (how many layers and components would have to be adapted or created), the cognitive complexity in respects to the layers and the flows of the application and the time needed to develop new features.

The implementation also revealed the complexity of each pattern in the number of lines written for each architecture (see column 6 of Table 2). We took MVC as a baseline, and we have observed that each architecture increases the number of lines of code. MVP showed an increase of 7.8%, MVVM 6.5%, VIPER 18.5%. While those numbers apply to our benchmark application, those could vary depending on the way the model and services layers are written, depending on the number of views and the overall complexity and features of the applications.

Increasing the granularity of the architecture automatically increases the number of layers and dependencies between components (see column 4 of Table

2). This not only introduces more code in order to link those but it also requires a higher level of skills from those who develop the application in order to be able to respect the overall architecture. The cost of granularity is more code written, more interfaces and more classes that are responsible for linking the layers.

The MVP architecture has shown an increase in the number of classes and interfaces from the base application (implemented with MVC). Based on the complexity of the UI elements for every initial View Controller we had at least one extra class and one extra interface, but this number can greatly vary based on the complexity of the application.

MVVM is very similar from this benchmark point of view with MVP; there was an increase of at least one class and one interface from baseline, but as in the case of MVP, this number considerable varies based on the complexity of the application and its architectural granularity. It is not uncommon in large enterprise projects to have more than 5 view-model classes for a View Controller.

In the case of VIPER, we have noticed that for every View Controller from the base application (implemented with MVC) the architecture required at least another 2 classes and 3 interfaces (protocols in Swift).

The number of dependencies increases with the complexity of the application and it is heavily influenced by the chosen architectural pattern. In the application we have benchmarked there were not so many complex views which could be implemented with the MVVM, however in applications with multiple view states and complex UI elements the percentage of code written and the number of dependencies would increase drastically from an MVC baseline point of view.

After the experiment, we have also calculated the Weighted Methods per Class (WMC) and the Coupling Between Objects (CBO) classes for the initial View Controllers of the MVC implemented application in order to reveal the complexity, reusability and the coupling of each architectural pattern. WMC counts the number of methods associated with a class, a high value indicates increased complexity and low reusability [3]. CBO it is a values which indicates the dependencies between classes, counting the relationships between classes without taking inheritance into account. A high CBO value indicates an increased dependency among classes and restricted reusability [3].

The codebases with the lowest WMC and CBO scores represented the ones which allowed a higher flexibility and testability, however they usually have a higher number of layers and components – which means higher development costs. It is important to mention that in the case of large and complex codebases those values are much more important than in the case of small and medium codebases as they accurately indicate the degree of flexibility, extensibility and testability. In smaller codebases, usually those metrics are not so important since the logic of the applications is less complex and the cognitive complexity of the flows is more easily to comprehend.

In the case of MVC, for one of the most complex View Controller of the application, the WMC was 28 and the CBO value was 2. MVP has shown a drastically decreased value of 14 for WMC and 1 for CBO for the same class, MVVM had its WMC value of 26 and the CBO 2. VIPER has as well shown a decrease in complexity with a WMC of 19 and a CBO value of 1.

The most testable architecture of them is VIPER as it provides great granularity and, with its concept of router classes, the navigation between views can also be unit tested. In the case of the other architectures the navigation between views is harder to test based on the way this is implemented (segues or programmatically modifying the navigation stack). MVVM provides increased testability for the user interaction components over MVC, while the MVP is as well more testable than MVC as presenters are usually plain objects and the interactions with those elements can be manually mocked and the events are not controlled by the iOS SDK.

4. Conclusions

We have analysed some of the most common software architectures used in mobile applications software development on a medium sized application on the iOS platform. The focus was on showcasing the strong and weak features of the evaluated architectural patterns on a real case example. We have considered the architectures from the MV family as these are the ones advocated for by the creators of the mobile operating systems, as well as different flavours of those. We have also included in our research VIPER, which is a relatively new architectural pattern that has gained a lot of popularity on the iOS mobile applications development scene.

The basis for our study was the implementation of the same iOS application with every one of the evaluated software architectures (MVC, MVP, MVVM, VIPER). After the implementation, we have examined each code base from the following points of view: flexibility, testability, dependencies between components and development costs.

After the experiment, we reinforced the assumption formed in years of commercially developing those kinds of applications regarding the importance of the software architectures. Software architecture has a critical role in the lifecycle of a mobile application and can strongly impact the cost of an application.

4.1. **Q1 - Why is the software architecture important in mobile applications?** For most companies that develop mobile applications, the cost is one of the most important factors in developing it. Having that in mind, choosing the right architecture for the application based on the functional requirements and the roadmap of the application can have a strong impact on the overall cost of the project.

For instance, it would not make sense for a proof of concept application (whose lifecycle is only a few months or a year) to be over-engineered. Spending time on implementing an architecture, which will provide flexibility for the future development of the application, has no sense from an economic point of view. By choosing a sophisticated architecture such as MVVM, the code base would increase dramatically and, based on the exact requirements of the application, the number of classes and line of codes could potentially double. In order to achieve those, the team which develops the application would have to be more skilled and the time for development and the cost will be directly proportional to the size of the codebase.

However not all mobile applications have the cost as one of the most important factors in their development. If we think about companies such as Snapchat, Tinder and Uber, all those companies are built around a mobile application and while they provide Web or Desktop applications as well, most of their revenue and user base comes from the mobile platforms. Those companies are not that concerned with the cost of the development and are more concerned with the extensibility of the application, its flexibility to adapt to new technologies, the range of devices on which the application can run, the ability to monitor the way their users interact with the application and to implement A/B testing for new features. They are also more concerned with the security and scalability of their application as well as the ability to provide new and interactive user interfaces and experiences and the ability to easily change these.

For mobile product companies, it makes a lot of sense and it is absolutely mandatory to have a software architecture which helps them achieve all their requirements. Failure to do so at the beginning of the project results in a technical debt which, most of the times, can only be leveraged by rewriting the application, or adding more resources and spending more time and money on the development.

Choosing the right software architecture for the product you are building while it is a hard task and, usually, involves people from all the layers of the company; product team, developers, business analysts etc., it is one of the most important task which you have to achieve and which will pay dividends in the long term. The importance of the task and the results of implementing it is closely related to the purpose of the application and its lifecycle. Nevertheless, it is important that at the beginning of the development of a product, after a thorough analysis, to make this decision.

4.2. **Q2: What does good software architecture mean in the context of mobile applications?** The purpose of the architecture is to provide a blueprint that is easy to follow and hard to break. It has to constrain the developer so that even the most inexperienced ones have to respect its principle and to avoid architectural erosion [11].

Good mobile software architecture should also be more flexible to change than other software architectures used in other types of software products such as web applications or embedded systems software. The reasons for this are the fact that mobile platforms are in continuous expansion and the field of the mobile application is reshaped every year with new kind of devices which have newer and more powerful hardware or they introduce completely new hardware which allows the developers to add unpredicted functionalities to their applications.

The most important aspect which heavily affects the architecture of an application is the development cost. The cost is influenced by the time of development, the skill of hired developers and the technology stack used. All these elements put their fingerprint on the final architecture of the product, that is why when starting a new project and choosing an architecture, it is really important to see if it is feasible from an economic point of view. Deciding for the wrong kind of architecture and not being able to correctly implement it while also delivering the required functionalities could cripple the project or badly erode the architecture in a way that might make it impossible to deliver new features or keep expanding it without massive refactoring.

4.3. **Q3: How can a mobile platform software architecture be analysed and benchmarked?** Mobile software architectures can be analysed as any other software architecture, from a complexity and granularity point of view, as well as for its flexibility and testability. What we have noticed is the fact that the mobile application architectures erode pretty fast as there are technological advancements in this field every year.

In order to benchmark an architecture, one has to first define the scope of the application, its lifecycle, release cycle and feature set. While there are many architectures that can be used for a certain application, its scope and budget usually dictate what architecture will be chosen. Given the budget and the scope of the application, we can filter the architectures and find out which one best fulfil the purpose.

Taking into consideration the scope of the application, the budget and the lifecycle, we can benchmark the architectures by their degree of flexibility, resistance to erosion, testability and ease of implementing.

Another important aspect in choosing the architecture is the number of its layers. While a greater granularity makes the whole architecture more flexible and testable and allows more developers to work on the same flow without creating conflicts, it also creates the need to write more code in order to link those components. Choosing the degree of granularity is as well as other features of an architecture strongly bounded to the purpose of the application heavily influences the lifecycle as well as the development costs.

The synchronicity between the model and what the user sees on the screen is another aspect that should be taken into consideration when choosing an architecture for a mobile application. There are architectures in which the view is automatically synchronised with the model the latter changes. Usually, the purpose of the application steers this kind of feature and synchronicity. For instance, a stock trading application would require that the model should always be in-sync with the view in order for users to see accurate prices. A photo browser application would not require this kind of synchronicity as it would not make sense from a network consumption and a user interaction point of view to refresh the feed of all the users when a new photo is added.

Nonetheless, choosing the right architecture is only the first step in building an application. The task of implementing an architecture is as important as choosing the right one and the skills of the development team usually affect the final product more than the chosen architecture.

4.4. **Future work.** As the next steps, based on this work, we plan to approach an auto-adaptation of an architectural pattern to the application; this approach should fit both medium sized applications and large, enterprise ones.

References

[1] Apple. Key value observing. https://developer.apple.com/library/archive/ documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html, 2018. Accessed date: 2018-06-30.

[2] S. Burbeck. Applications programming in smalltalk-80 (tm): How to use model-view-controller (MVC). *Smalltalk-80 v2*, 5:1–11, 1992.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

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

[5] J. Gilbert and C. Stoll. Architecting iOS apps with VIPER. https://www.objc.io/issues/13-architecture/viper/, 2014. Accessed date: 2018-04-02.

[6] S. Khanlou. Massive View Controller. http://khanlou.com/2015/12/massive-view-controller/, 2015. Accessed date: 2018-06-30.

[7] H. J. La and S. D. Kim. Balanced MVC architecture for developing service-based mobile applications. In *e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on*, pages 292–299. IEEE, 2010.

[8] D. Mark, J. LaMarche, and J. Nutting. *More iPhone 3 Development.* Springer, 2010.

[9] R. C. Martin. *Clean architecture: a craftsman's guide to software structure and design.* Prentice Hall Press, 2017.

[10] MutualMobile. Meet VIPER: Mutual mobile's application of clean architecture for iOS apps. https://mutualmobile.com/posts/meet-viper-fast-agile-non-lethal-ios-architecture-framework, 2014. Accessed date: 2018-04-02.

[11] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes*, 17(4):40–52, 1992.

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

[13] M. Potel. MVP: Model-view-presenter the taligent programming model for C++ and Java. *Taligent Inc*, page 20, 1996.

[14] T. Reenskaug. The model-view-controller (MVC): its past and present. *University of Oslo Draft*, 2003.

[15] T. M. H. Reenskaug. The original MVC reports. Technical report, Xerox Palo Alto Research Laboratory, PARC, 1979.

[16] F. J. A. Salazar and M. Brambilla. Tailoring software architecture concepts and process for mobile application development. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 21–24. ACM, 2015.

[17] P. Sauter, G. Vögler, G. Specht, and T. Flor. A MVC extension for pervasive multi-client user interfaces. *Personal and Ubiquitous Computing*, 9(2):100–107, 2005.

[18] M. A. Sayed. VIPER design pattern for iOS application development. https://medium.com/@smalam119/viper-design-pattern-for-ios-application-development-7a9703902af6, 2017. Accessed date: 2018-04-02.

[19] F. E. Shahbudin and F.-F. Chua. Design patterns for developing high efficiency mobile application. *Journal of Information Technology & Software Engineering*, 3(3):1, 2013.

[20] A. Sinhal. MVC, MVP and MVVM design pattern. https://medium.com/@ankit.sinhal/mvc-mvp-and-mvvm-design-pattern-6e169567bbad, 2017. Accessed date: 2018-04-02.

[21] K. Sokolova, M. Lemercier, and L. Garcia. Android passive MVC: a novel architecture model for android application development. In *International Conference on Pervasive Patterns and Applications*, pages 7–12, 2013.

[22] K. Sokolova, M. Lemercier, and L. Garcia. Towards high quality mobile applications: Android passive MVC architecture. *International Journal On Advances in Software*, 7(2):123–138, 2014.

[23] Statcounter. Mobile operating system market share worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide/monthly-201705-201805, 2018. Accessed date: 2018-06-30.

[24] C. Trevino. Flux and presentation architectures. https://blog.atsid.com/flux-and-presentation-architectures-91283f7ef94b, 2016. Accessed date: 2018-04-02.

Babeş-Bolyai University, Department of Computer Science, 1 M. Kogălniceanu Street, 400084 Cluj-Napoca, Romania

Email address: `dobrean@cs.ubbcluj.ro, lauras@cs.ubbcluj.ro`