Nigerian Research Journal of Engineering and Environmental Sciences 6(2) 2021 pp. 607-617
p ISSN: 2635-3342; e ISSN: 2635-3350

Nigerian Research Journal of Engineering and Environmental Sciences
Journal homepage: www.rjees.com

Original Research Article

Android Architectures for a Robust Mobile Application Development

[1]Audu, J.B., *[2]Dahunsi, F.M., [3]Obe, O.O. and [3]Sarumi, O.A.

[1]Department of Electrical and Electronics Engineering, Federal University of Technology, Akure, Ondo State, Nigeria.
[2]Department of Computer Engineering, Federal University of Technology, Akure, Ondo State, Nigeria.
[3]Department of Computer Science, Federal University of Technology, Akure, Ondo State, Nigeria.
*fmdahunsi@futa.edu.ng

http://doi.org/10.5281/zenodo.5805197

ARTICLE INFORMATION

Article history:
Received 26 Nov 2021
Revised 10 Dec 2021
Accepted 12 Dec 2021
Available online 30 Dec 2021

Keywords:
Android Architecture
Model-view-view-model
Model-view-presenter
Model-view-controller
Quality of service

ABSTRACT

Over the past ten years, mobile phone usage has been on a tremendous increase globally. The two top leaders of the mobile market are Android by Google and iOS by Apple. The Android OS has witnessed great popularity in Africa, especially Nigeria, with 41.14 million Android users. Research shows at least five other similar applications for every single application out of the total 2.87 million applications on the Google Play store. Hence, it must be cost-effective and good quality for every mobile application released to be competitive. An Android application's quality and development time directly depend on the architecture choice. Therefore, this paper aims to compare the two major Android architectures model-view-view-model vs model-view-presenter (MVVM vs MVP) with the model view controller (MVC) architecture as a benchmark using different approaches to propose the most suitable architecture for developing a mobile application with specific requirements for a user participatory mobile quality of service analysis. Findings from this paper show that the optimal choice to make for the development of mobile quality of service (QoS) application is the MVVM architecture. The MVVM architecture showed better results than the MVP architecture regarding evaluated metrics (testability, modifiability, and performance).

1. INTRODUCTION

One crucial step in producing high-quality software is developing a software architectural model (Schmerl et al., 2006). In the bid to develop a good quality of service (QoS) service application (App), many factors

need to be considered, including App performance, code maintainability, and code flexibility. All these factors are greatly influenced by the architectural design pattern of the application. There are three significant architectures used in the development of an Android application which include the model view controller (MVP), model view view-model (MVVM), and model view presenter (MVP) (Wisnuadhi et al., 2020). Research has shown that the MVVM and MVP architectures are better for development than the MVC architecture, despite MVC being the most popular due to implementation ease. Much previous work has compared the MVVM and MVP architecture. For example, Wisnuadhi et al. (2020) compared the two architectures in terms of performance. Other authors compared the MVVM and MVP architectures based on testability, modifiability, and performance criteria, focusing on the iOS platform (Sholichin et al., 2019). Vladyslav (2018) evaluated testability, modifiability, maintainability, and performance; they compared the MVP architecture and the View Interactor Presenter Entity Router (VIPER) architecture.

The Android operating system has a complex structure (Muntenescu, 2016; Vladyslav, 2018). When developing Android applications, architecture has been a subject of great interest. Several researchers have suggested various reasons for selecting one architecture over the other. Sokolova et al. (2004) investigated different architectural design patterns, then proposed and demonstrated the implementation of the MVC design pattern with a sample application. Vladyslav (2018) compared the MVP and the VIPER architecture. He evaluated both based on testability, modifiability, maintainability, and performance. The research outcome showed that VIPER architecture has a better performance than the MVP. Lou (2016) compared the MVP, MVC, and MVVM architecture using a scenario-based approach to evaluate the testability and modifiability and a simulation-based process to evaluate the performance of these architectures. The work's outcome showed that MVVM was better than MVP, which was better than MVC in testability. MVVM is equal to MVP but better than MVC in terms of modifiability, while MVP is greater than MVVM, which was more significant than MVC in performance.

This work differs from other similar works because, in previous works, the difference between MVP and MVVM architectures in Android is still unclear as comparisons are only made based on one or two criteria. However, this work compares the MVP and MVVM architecture based on three criteria: testability, modifiability, and performance criteria, and focused mainly on the Android platform.

This paper compares the MVVM and MVP design patterns using the MVC as the benchmark by performing an in-depth analysis of the architectures using a homegrown mobile quiz App. To evaluate the best architecture that meets the quality requirement of an Android application for a user participatory mobile quality of service analysis.

2. METHODOLOGY

2.1. Design Patterns Implementation in Android

A sample Quiz Android application was developed to demonstrate the three architectures namely MVC, MVP, and MVVM. The Quiz App was selected because it is simple yet complex enough to distinguish between the three evaluated architectures, and the User interface is presented in Figure 1.

2.1.1. MVC implementation in Android

Figure 2 describes the implementation of the MVC architecture in Android with the help of a simple Quiz-App. Figure 2 shows the View, which combines the layout and the activity. The layout resource does not have complete control over the View (Lou, 2016). The View must be inflated in the MainActivity using the setContentView() method shown in the Controller. The Controller is responsible for setting up the views. It retrieves the question from the model and displays it to the user via the View. It receives an event from the user, verifies whether the answer is correct with the logic implemented in the controller class, notifies the user whether the solution chosen is correct, and finally queries the model for the following question data. The model is a data class in Kotlin (equivalent to a POJO class in JAVA) that handles the logic to get the next question.

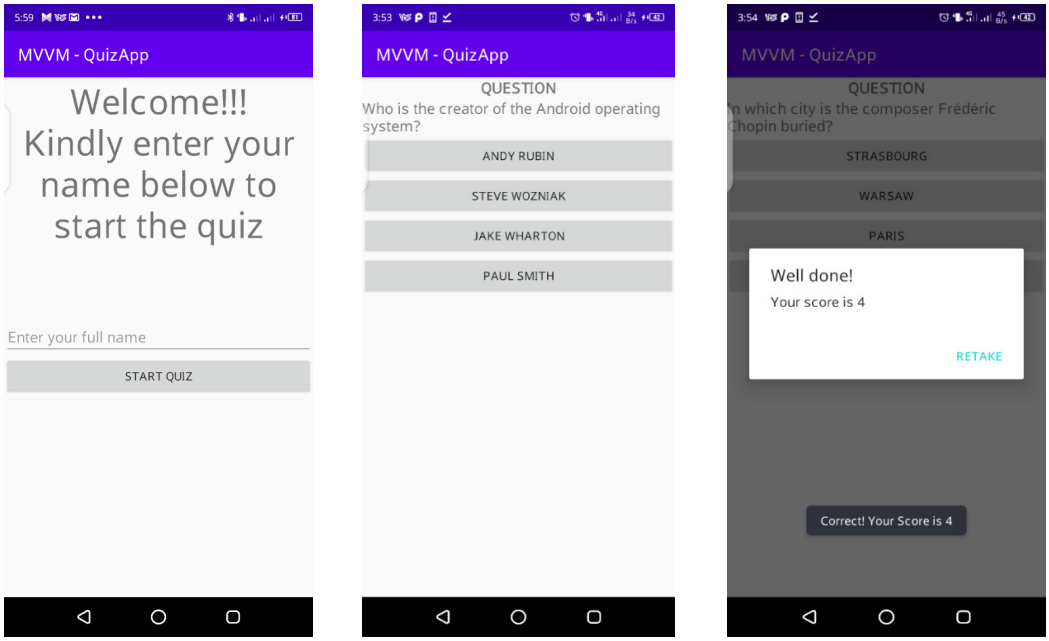

Figure 1: The quiz app user interface

2.1.2. MVVM implementation in Android

In Android, the MVVM design architecture functions effectively with a specific binding technology (Data binding library). The data binding library was introduced during Google I/O 2015, which helps write declarative user interface (UI) and minimize glue code necessary to bind application logic and layouts (Husayn, 2017). There is only a single layer of connection between each component except the Repository connected to the local and remote data source as illustrated in Figure 3. The Repository simply unifies the two data sources into one reference to supply the View-model. Thus, the View-model has no idea whether to get remote or local data.

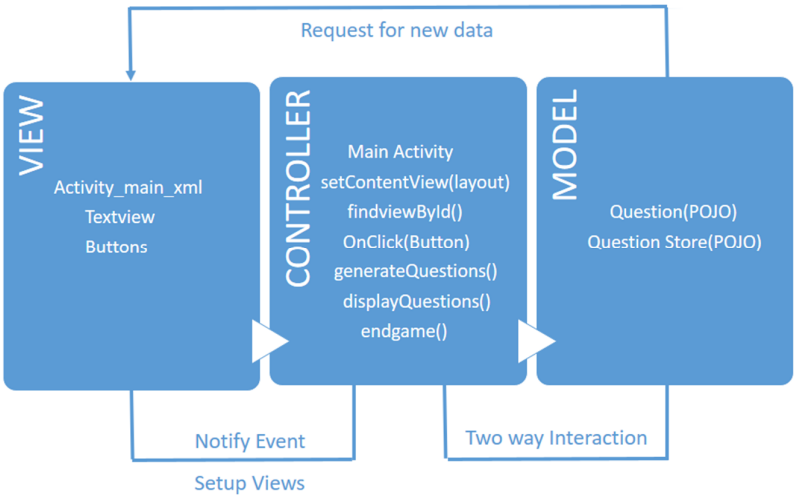

Figure 2: Model view controller Android implementation

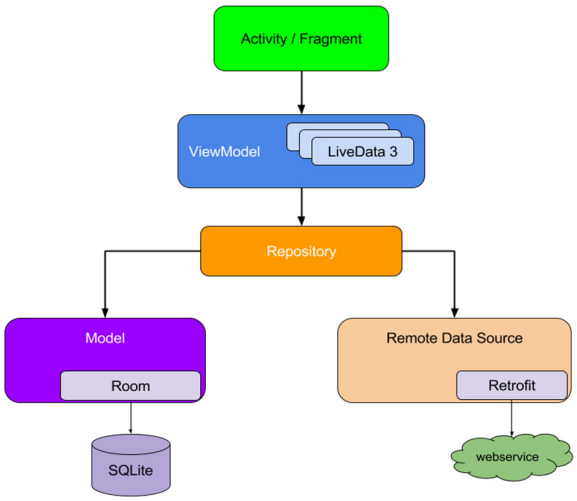

Figure 3: Model view view-model architecture structure

The MVVM architecture was demonstrated with the same Quiz-App used in presenting the MVP architecture. The MVVM implementation in Figure 4 distinctively portrays that the View does not contain any logic. It is only responsible for initializing the binding components and the View models. Though not depicted in Figure 4, the view-model directly binds with the view component in the layout file. An Observable (LiveData in this case) wraps each data used by the View to ensure that the view updates automatically when the data changes.

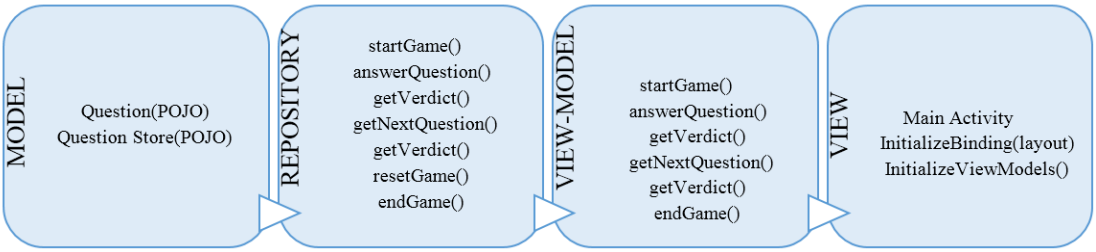

Figure 4: Model view view-model Android implementation

The model class only describes the object attributes. The View-Model class holds all the necessary logic ranging from starting the game, presenting the next question, verifying the answer, to ending the game. The view-model abstracts the Repository from the View and directly calls most of the methods implemented in the Repository necessary for the View to function. Data changes based on user action, such as the question (which holds the questions and corresponding options at a particular point during the game). The Verdict (which changes based on the user-chosen option) and the LiveData Observable wrap the game's status (whether the quiz has ended or not). It quickly allows notification of the View directly when changes are made to this data instead of notifying it via code. The implementation shows that the User Interface is completely decoupled from the application's logic.

2.1.3. MVP implementation in Android

The MVP design has a different flavour of implementation in Android, which is demonstrated here with the performance of the developed Quiz-App. Figure 5 presents the View implemented with an activity (TakeQuizActivity) and only contains a reference to the Presenter. The View function only calls the presenter method (e.g startGame() method) whenever the user acts. Some interfaces (e.g., showVerdict-Message() method) defined in the presenter class are overridden by the TakeQuizActivity class. These listeners trigger

an action to automatically update the views whenever the state of the data changes. The Presenter acts as an intermediary between the View and the model (Bachmann et al., 2007; Leiva 2014). It implements the interactor listener and overrides all the methods required to change the View state when changes occur in the model. The Presenter prepares the method for each possible action the View can perform. (e.g .showNewQuestion(), showQuizResult(), showVerdictMessage()). The View is wholly abstracted from the model, and the components are not also tightly coupled and show a great separation of concern. All logics are implemented in the presenter class, while the View is specifically responsible for displaying the data presented by the Presenter.

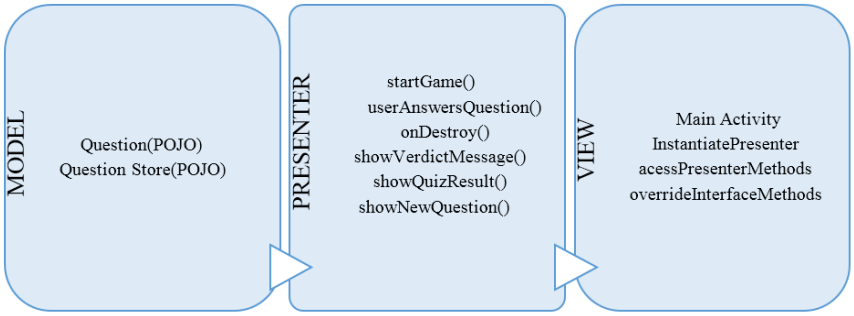

Figure 5: Model-view-presenter Android implementation

2.2. Comparative Analysis of Implemented Design Patterns

This section describes the metrics used in comparing the three architectural patterns. The actual sample Android app was carried out separately on the three different architectural patterns. It is demonstrated in detail how the metrics varied across the other architectures. The comparison between the three architectural patterns was investigated based on three metrics; Testability, Modifiability, and Performance (Kazman et al., 2000; Mattsson et al., 2016).

2.2.1. Testability

Android Developer documentation (Google Inc) stated that testing an app is an integral part of the app development. It allows developers to verify the developed application's correctness, functional behaviour, and usability before release. There are two major test types written for android applications: local unit tests and Instrumented tests. The local unit tests run on the Java Machine independent of the Android framework or written with mocked Android dependencies. While the instrumented tests are Android platform-dependent, they run on an actual hardware device or emulator. According to Mobile app performance (2020), these instrumented tests are written when writing integration and functional UI tests to automate user Interaction. Alternatively, the test has Android dependencies that mock objects cannot satisfy.

This section presents significant criteria to consider when measuring the testability of an Android Application. Lou (2016) discussed extensively why these criteria are selected and used to measure the testability metric. The paper explains how previous researchers have argued and presented the listed criteria as suitable for measuring the testability of design architecture. This work focuses on four primary testability criteria, detailed as follows:

i. Ease of application test
ii. Size of the test cases
iii. Consumed time running test
iv. Ease of debugging

The selection of these four criteria was based on the similarity irrespective of the architecture in the process. Regarding the ease of interpreting test results and locating errors. Observability and controllability are dependent on the function details, which do not change when components are decoupled across architectures;

this makes it also stay the same across different architectures. From the architectural description of the three design patterns, it is clear that the MVVM and MVP architectures are comparatively better than the MVC architecture; hence the MVC architecture was used as a benchmark to compare the MVP and MVVM architecture.

2.1.1.1. Comparing MVVM and MVC

Ease of application test: Figure 2 shows that the application logic is written in the controller class hosted by the View's same activity hosts. This implies that every method handling the tests' logic will require many mocked Android dependencies. On the other hand, the MVVM design pattern shows a clear separation of concern. The Views are entirely decoupled from the model, and a single Repository class handles the logic, enhancing effective and efficient testing.

Table 1: Summary of comparison of MVVM and MVP using MVC as a benchmark

Testability criteria	MVVM vs MVC	MVP vs MVC
Ease to test the application	• Enhances effective and efficient testing	• Enhances effective and efficient testing
Size of test cases	• Sizes of test cases are significantly smaller	• The size of test cases are the same
Consumed time to run test	• Less time to complete tests	• Less time to complete tests
Ease to debug	• Easy to find a bug as long as the associated component is known • Debugging with breakpoints is not favourable	• Easy to debug • Debugging with breakpoints is less favourable

Size of the test cases: Although this sample application is not complex enough to show the significant difference between the size of the Test-cases, it is inevitable that as the complexity of the application grows. The methods defined in the MVVM pattern will be reusable, avoiding extra costs methods that will increase the test cases. Hence, the size of the Test-cases with the MVVM design architecture will be significantly smaller than the Test-cases' size with the MVC design pattern.

Consumed time running tests: As aforementioned, all of the logic method from the View in the MVC design pattern was moved to the Repository class in the MVVM pattern allowing for more local unit tests and less instrumented tests. As the local unit tests run on the Java Virtual Machine (JVM), it will consume less time to complete compared to instrumented tests that would need to spin up the Android emulator to run tests.

Ease of debugging: Although debugging with breakpoints (a built-in feature offered by Android studio) in the MVC design pattern is straightforward. The logic associated with a particular view is in a single file, so it is easy to find a bug compared to the MVVM pattern in which the logic has usage and reference in multiple files. Furthermore, the MVVM is at a disadvantage when debugging with breakpoints as layouts do not support setting breakpoints to debug. However, the high separation of concern of the MVVM architecture cannot be overlooked, as it is easy to find a bug once the component is associated with is known.

2.1.1.2. Comparing MVP and MVC

Ease of application test: The application logic is written in the controller class hosted by the same activity that hosts the View, as presented in Figure 2. This shows that every method that handles logic that tests would be written for will require that many Android dependencies be mocked, provided that the mock objects can satisfy these tests even. The MVP design pattern, like the MVVM shows a clear separation of concern. The views are entirely decoupled from the model, and the presenter class handles the logic enhancing effective and efficient testing.

Size of the test cases: The same method in the MainActivity in the MVC implementation was moved to the presenter class in the MVP implementation. Since the two implementations have the same functionality, the size of the test cases is the same.

Consumed time to run tests: All of the logic methods from the View in the MVC design pattern were moved to the Presenter class in the MVP pattern allowing for more Local unit tests and less instrumented tests written. As the local unit tests run on the Java Virtual Machine (JVM), it will consume less time to complete compared to tests that would need to spin up the Android emulator to run tests.

Ease of debugging: Although debugging with breakpoints (a built-in feature offered by Android studio) in the MVC design pattern is straightforward, the logic associated with a particular view is in a single file, so it is easy to find a bug. It is also easy to debug with the MVP pattern but less convenient because the method in a single class in the MVC design is split into multiple classes in the MVP implementation.

2.2.2. Modifiability

Modifiability can be regarded as the cost to make changes to an application, the number of classes added to implement a new feature or the improvement of an existing feature. It defines the number of classes that need editing, the number of methods added, the total number of lines of code added to implement or modify a feature. This paper employed the same simulation-based approach as Vladyslav (2018) to evaluate the modifiability of these various architectures. The two main criteria to measure the modifiability were:

i. The number of classes to implement a new feature
ii. The number of lines of code added or deleted to implement or modify existing features

Table 2 shows that the total number of classes created in the MVP app is 8, while 10 and 11 were created in the MVVM App and MVC App, respectively. Both the MVP and MVVM have more classes than the MVC App. They show that responsibilities are shared between many classes, with each having a small set of responsibilities. From Tables 2 and 3, the total additional lines of code (loc) added to the MVC App to implement a new feature was 377-222=155 loc. The 457-314=143 line of code was added to implement the new feature in the MVP App. At the same time, 393-256 =137 lines of code were added to implement the new feature in the MVVM app. In the MVC, fewer lines of code were written to implement a new feature compared to MVP and MVVM. This shows that modifying the application to implement a new feature will be faster in MVC than in other architectures.

Table 2: Number of classes initially created

Feature	MVC	MVP	MVVM
Domain layer	3 classes	3 classes	3 classes
Base architecture	0 classes	3 classes	2 classes
Add information	1 class	1 class	1 class
Take quiz	1 class	1 class	1 class
Save result	1 class	1 class	1 class
Send result to server	2 classes	2 classes	2 classes
Total	8 classes	11 classes	10 classes

Table 3: Number of lines of code before implementing a new feature

Feature	MVC	MVP	MVVM
Domain layer	28 loc	28 loc	28 loc
Base architecture	0 loc	187loc	171loc
Take quiz	194loc	99loc	57loc
Total	222	314	256

Table 4: Number of lines of code after adding a new feature

Feature	MVC	MVP	MVVM
Domain layer	31 loc	31 loc	31 loc
Base architecture	0 loc	213 loc	201loc
Add user information	37 loc	36 loc	25 loc
Take quiz	239 loc	107 loc	66 loc
Send result to server	70 loc	70 loc	70 loc
Total	377	457	393

2.2.3. Performance

Performance is how an application meets its required specification within the given constraints (Lou, 2016). Research shows that about 50% of mobile App users have experienced performance problems (Google Inc.). According to Alberto and Giancarlo, (2012), factors considered when evaluating the performance of a software include; memory, execution time, memory usage, and battery consumption. Based on previous research work and as suggested on the Android developer website, the following criteria are considered when evaluating the performance of an application (Alberto and Giancarlo, 2012; Wassenaar, 2017; Vladyslav, 2018):

i. CPU usage
ii. Memory usage
iii. Network usage
iv. Battery usage
v. Execution time

In this paper, the performance evaluation of the different design patterns was not scenario-based as in the testability case. Instead, it was simulation-based, and three performance criteria were measured for each architecture. For this evaluation, the Android profiler measures the various performance of the different design patterns implemented with the same QuizApp.

3. RESULTS AND DISCUSSION

3.1. Performance evaluation

The simulation was carried out by automating user action using espresso (a testing framework for Android to make it easy to write reliable user interface tests) (Vogel, 2016). The basic functionality of the App is to display questions to the user. After the user clicks an option (a button), the user's answer is verified, and a new question is displayed to the user. After the user answers all the questions, the result is saved to the file, including the time information. A background job that mimics uploading the result file to a server is scheduled to run immediately in the background.

The espresso framework automates user action (clicking the buttons to answer questions). The number of questions to be displayed was increased to 100, thus making the espresso frame automate user clicking 100 times in approximately three minutes. While carrying out this test, the android profiler is activated to measure the performance metrics. The Android profiler calculated the performance metrics in real-time, but the measurement was considered at an interval of 10 seconds until the final minute.

Table 5 shows the data collected for each performance criteria over two minutes, within which a user action was performed repeatedly by the espresso framework and the calculated average performance within these two minutes.

Table 5: Measured CPU usage by each application

	Measured CPU usage												
Design	10	20	30	40	50	60	70	80	90	100	110	120	Mean
MVC	3	1	11	1	3	8	2	3	15	3	2	3	4.58
MVP	0	1	5	1	1	5	1	0	3	1	11	1	2.50
MVVM	1	9	1	0	5	1	1	0	1	1	1	6	2.25

The average CPU usage over the period by the MVC App was 4.58%, 2.5% for the MVP App, and 2.25% for the MVVM App, as shown in Table 5. The minimum usage by the MVC App was 1%, and the maximum was 15%, while the minimum for the MVP App was 0%, and the maximum was 11%. On the other hand, the MVVM App has a maximum of 9% and 0%. The MVVM and the MVP app use fewer CPU resources than the MVC app. Nevertheless, the MVVM app uses comparably fewer resources than the MVP app.

The MVC app's average memory usage overtime was 73.09%, 44.2% for the MVP App, and 44.68% for the MVVM app, as presented in Table 6. The minimum memory usage by the MVC app was 64.12%, and the maximum was 83.8%, while the minimum for the MVP App is 39.9%, and the maximum is 50.5%. The MVVM App, in comparison, had a maximum of 51.3% and a minimum of 38.9%. The MVVM and the MVP required less memory allocation than the MVC app. However, the MVVM App uses less memory compared to the MVP app. Generally, aside from the MVC App that hits a medium value at a particular point, the three architectures in terms of battery consumption do not significantly differ, as shown in Table 7. Considering these three criteria, the MVVM architecture is better than the MVP in performance.

Table 6: Memory usage by each application

	Memory usage												
Design	10	20	30	40	50	60	70	80	90	100	110	120	Mean
MVC	64.1	64.1	65.6	68.9	71.2	72.8	74.9	76.8	78.6	81.3	83.8	75.0	73.09
MVP	41.9	44.4	41.8	43.6	45.7	39.9	41.1	42.5	44.4	46.3	48.3	50.5	44.20
MVVM	38.9	41.2	43.0	39.4	41.4	43.1	44.9	46.9	48.9	47.6	49.5	51.3	44.68

Table 7: Measured energy usage by each application

	Energy usage											
Desig	10	20	30	40	50	60	70	80	90	100	110	120
MVC	medium	Light	light	light	light	light	light	light	light	light	medium	light
MVP	light	Light	light	light	light	light	light	light	light	light	light	light
MVV	light	Light	light	light	light	light	light	light	light	light	light	light

3.2. Metrics Comparison

Comparing the three architectures, modifications, testability, and performance metrics were selected.

3.2.1. Testability criteria

Four criteria were considered when measuring the testability of each architecture.

Criteria one: ease of application test

The MVVM pattern enhances effective and efficient testing as it shows a clear separation of concern compared to the MVC architecture. The MVP architecture is also straightforward to test if the View is completely decoupled from the model and a separate class handles the logic.

Criteria two: the size of test-cases

The size of test cases with the MVVM architecture grows significantly smaller as the application grows, while the size of Test-cases of the MVP pattern remains; this is the same for the MVC pattern.

Criteria three: consumed time to run tests

Most of the instrumented test in the MVC architecture is converted to unit tests in the MVVM and MVP pattern. As it takes less time to run unit tests than instrumented tests, the consumed time to run tests for the MVVM and MVP is better than that of MVC.

Criteria four: ease of debugging

It was observed that debugging with the breakpoint in the MVVM architecture can be challenging compared to the MVP and MVC architecture. In terms of "ease of test application", the MVVM and MVP architecture are equal. In terms of "Size of test-cases" and "Consumed time to run tests," the MVVM architecture outperforms the MVP architecture, while in terms of "Ease to debug," the MVP architecture is better. MVVM architecture is better than the MVP architecture in terms of testability, from the test carried out was deduced that the.

3.2.2. Modifiability criteria

The second comparison metric considered is modifiability. The criteria considered in evaluating the modifiability are:

Criteria one: number of classes added to implement a new feature

The evaluation result displayed in Table 1 shows that both the MVVM and MVP patterns differ in the number of classes and are higher than those in the MVC pattern. This indicates that responsibilities are shared between a higher number of classes. They are enabling each class to handle small responsibilities.

Criteria two: number of lines of code to implement or modify an existing feature

The MVVM architecture required fewer lines of code to implement a new feature. Combining these two criteria shows that the MVVM architecture allows for easier modifiability than the MVP architecture.

3.2.3. Performance criteria

The performance criteria considered are:

Criteria one: memory usage

The MVP has better memory usage, using 0.48Mb less than the MVVM architecture.

Criteria two: CPU usage

The MVVM architecture uses 0.25% fewer CPU resources than the MVP architecture.

Criteria three: battery consumption

The three architectures, comparatively, do not consume too much energy, though the MVVM performed much better amidst the three architectures.

As presented above, the MVP and MVVM architecture are far better than the MVC architecture in all three metrics. On the other hand, Comparing the MVVM and MVP architectures, architectures' testability was evaluated based on three criteria. The MVVM architecture performed better in two of the four criteria considered. The MVP architecture performed better in one, while both performed equally in the remaining criteria.

Furthermore, the modifiability was evaluated based on two criteria: the MVP architecture performed better in terms of "number of classes," and the MVVM architecture performed better in "lines of code that needed to implement a new feature."

Finally, a simulation to evaluate the performance of each architecture based on three criteria was implemented: (1) Memory Usage, (2) Central Processing Unit (CPU) Usage, and (3) Battery Consumption. Both the MVVM and MVP architecture performed better than the MVC architecture. The MVVM architecture beats MVP architecture in two of the three criteria considered.

4. CONCLUSION

This paper compared two major android development architectures (MVVM and MVP) using the MVC architecture as a benchmark to make a good choice of architecture for the QOS Mobile application. A test application was developed, and the three architectures (MVVM, MVP, MVC) were evaluated based on testability, modifiability, and performance. The result obtained shows a close competition between the MVP and MVVM architecture, where one is better in one metric than the other and vice-versa. The two architectures are viable. As much as any of the two architectures will be suitable for developing a user participatory mobile quality of service analysis application, selecting the best architecture is made based on the application requirement.

5. ACKNOWLEDGMENT

This research was funded by the Nigerian Communication Commission Research Fund Grant 2020

6. CONFLICT OF INTEREST

There is no conflict of interest associated with this work.

REFERENCES

Alberto, S. and Giancarlo, S. (2012). Mobile multi-platform development: An experiment for performance analysis. The 3rd International Conference on Ambient Systems, Networks and Technologies (ANT-2012), Canada International Conference on Mobile Web Information Systems (MobiWIS 2012), Niagara Falls, March 2012, In: Procedia Computer Science 10(2012) pp. 736 – 743.

Bachmann, F., Bass, L. and Nord, R. (2007). *Modifiability tactics*. Pittsburgh: Carnegie Mellon University Pittsburgh Pa Software Engineering Inst.

Husayn, H. (2017). Android by example: MVVM +Data Binding -> Introduction (Part 1). Retrieved May 21, 2020, from https://medium.com/@husayn.hakeem/android-by-example-mvvm-data-binding-introduction-part-1-6a7a5f388bf7

Kazman, R., Klein, M. and Clements, P. (2000). ATAM: Method for architecture evaluation. Pittsburgh: Carnegie-Mellon University Pittsburgh PA Software Engineering Inst.

Leiva, A. (2014). Model View Presenter in Android. Retrieved April, 2020, from https://antonioleiva.com/mvp-android/

Lou, T. (2016). A comparison of Android Native App Architecture MVC, MVP and MVVM. Eimdhoven University of Technology, Eimdhoven.

Mattsson, M., Grahn, H. and Mårtensson, F. (2016). Software architecture evaluation methods for performance, maintainability, testability, and portability. Second International Conference on the Quality of Software Architectures, Västeras, Schweden, June 27-29, 2016.

Mobile App Performance. (2020). Retrieved May 2020 from Jmango: http://www.jmango.360.com/blog/how-to-improve-your-mobile-app-performance/

Muntenescu, F. (2016). Android Architecture Design Patterns. Retrieved April 2020, 1, from https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6

Schmerl, B., Buttler, S. and Garlan, D. (2006). *Architecture-based Simulation for Security and Performance*. Pittsburgh, USA.: School of Computer Science, Carnegie Mellon University.

Sholichin, F., Adha, I. M., Halim, S. and Firdaus, M. (2019). Review of iOS Architectural Pattern For Testability, Modifiability, and Performance Quality. *Journal of Theoretical and Applied Information Technology, vol. 97, No. 15, pp. 4021-4035*.

Sokolova, K., Lemercier, M. and Garcia, L. (2004). Towards High-Quality Mobile Applications: Android Passive MVC Architecture. *International Journal on Advances in Software*, 7(15), pp. 123 – 138.

Vladyslav, H. (2018). 'Android Architecture Comparison: MVP vs VIPER', Linne: PhD dissertation, Computer Science Department, University of Linneuniversitetet.

Vogel, L. (2016). Android User Interface Testing with Espresso. Retrieved May 6, 2020, from https://www.vogella.com/tutorials/AndroidTestingEspresso/article.html

Wassenaar, D. (2017). How to Improve your Mobile App Performance. Retrieved May 1, 2020, from https://jmango360.com/blog/how-to-improve-your-mobile-app-performance/

Wisnuadhi, B., Munawar, G. and Wahyu, U. (2020). Performance Comparison of Native Android. International Seminar of Science and Applied Technology in Advances in Engineering Research, volume 198. Published by Atlantis Press B.V..