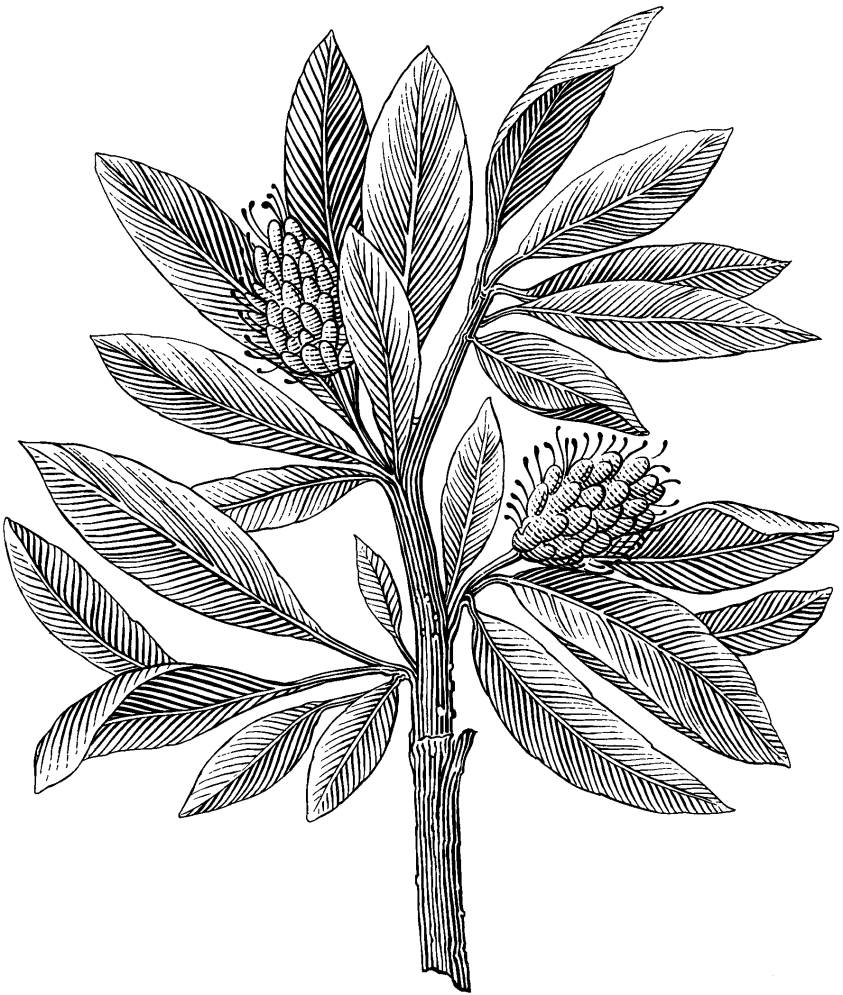
Linnéuniversitetet
Kalmar Växjö

Master Thesis Project

Android Architecture Comparison: MVP vs. VIPER



Author: Vladyslav Humeniuk
Supervisor: Welf Lowe
Examiner: Andreas Kerren
Reader: Mattias Davidsson
Semester: VT 2018
Course Code: 4DV50E
Subject: Computer Science

Abstract

Android application development has been of interest since first Android smartphone was released. Applications are constantly getting more complex as well as smartphone hardware is getting better. New ways of developing android applications are developed with time. There is Model View Presenter architecture that is the most used for android applications now and new View Interactor Presenter Entity Router architecture that is becoming more popular. But there is no empirical data to compare these architectures to understand what architecture will fit better for developing new applications. This thesis aims to compare the MVP and the VIPER android architectures using a few important metrics like maintainability, modifiability, testability and performance. Results will answer what architecture is better for developing different types of projects. VIPER architecture showed better performance results and maintanance metrics comparison show that both architectures have advantages and disadvantages.

Keywords: software architectures, mobile development, android, mvp, viper, architecture comparison.

Contents

1 Introduction

Every application is built using a specific architecture. Different architectures offer different advantages that come with different weaknesses. Application architecture choice often depends on the specific requirements of the project and properties of its environment. Software architectures usually have different properties to fit different needs. Some architectures show better performance on big projects, but require much more time to be implemented. Other architectures provide great versatility for a cost of code structure, that is harder to cover with tests.

Software architecture design is the field of knowledge that was constantly developed in the last ten years to improve quality of software systems and to find architectural solutions for new challenges [17]. Architectural knowledge will be even more important in future because new systems require new approaches for development. New software architectures are introduced constantly. These architectures are designed to solve problems of existing architectures. But the understanding of advantages and disadvantages of the new architecture compared to the more conservative one should be evaluated by scientific research. Comparing two architectures requires a method to be invented. Architecture is usually described by a set of properties, like modifiability or performance. Methods, that are designed to compare architectures, usually rely on comparison of these properties. However, simply comparing properties is not enough. The results of the properties evaluation should be analyzed and explained in a way, that can actually be used to make a decision on architecture selection for a specific project.

1.1 Background

Bob Martin described base principles of Clean Architecture in his book “Clean Code” [8] Architecture should separate layers to increase maintainability and follow SOLID principles. SOLID is an acronym of names of principles. These are Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle. The main purpose of these principles is to make software more understandable, maintainable and flexible.

The android operating system has a complex structure. Combined with specifics of how applications should be built, it adds some restrictions to the application architecture.

The android operating system is built upon few major components and layers. This is described in more details in section 2.1 Android platform. The top layer is application layer. All applications are executed on this level. To access operating system functions, applications use the API of the Application Framework layer. The application framework is responsible for providing device and system data via an API to applications. It collects data from different sensors and hardware. For this purpose, it uses system libraries and the Android Runtime environment.

Android system libraries and the Android Runtime environment are on the same layer and responsible for providing basic functions. These libraries provide access to hardware, for example, light sensors or accelerometer. They provide classes to enable media control, like audio or video playback. They give access to interfaces for displaying text on the screen, drawing graphics. The most basic

layer is the Linux kernel. It is the core of the operating system. It contains drivers of hardware components and operates with those.

1.2 Motivation

The Android operating system is the most popular mobile development platform with around 80% market shares today [13]. Hence, research in this field of software development will be beneficial for lots of developers. New Android applications are constantly being released on the Play Market.

Android applications are now being developed mostly using MVP architecture. But there is a new architecture VIPER, coming from iOS development. It is being adapted for Android development. Both these architectures are used to follow Bob Martin's advice for building clean architecture for mobile projects. Bob Martin introduced layer separation for architecture and declared base principles and responsibilities for each layer. Layers are following:

- Data access layer. Classes in this layer are responsible for storing and retrieving domain objects.
- Business logic layer. Classes in this layer are responsible for any logical operation on the domain object.
- Presentation layer. Classes in this layer are responsible for creating user interface visible to users.

Classes in presentation layer should only access business logic layer and should not know about data access layer. Classes in business login layer should only operate using data access layer. The data access layer should be independent of any other layer. Application architecture has a high influence on the overall software quality. Architecture is called one of the significant requirements for software quality [10].

There are not many articles describing VIPER architecture and there is no empirical data and thorough analysis to understand what architecture is more suitable for different projects with different specifics. Developers still doubt if projects should be migrated to the new architecture and what advantages they get. The thesis aims to fill the research gap of VIPER architecture and comparison it with more traditional MVP architecture.

1.3 Problem Statement

Software systems and applications properties are highly dependent on the architectural approach they are built on. Because of this, choosing the right architecture to fulfill all the needs is a highly important task. Rebuilding an application always takes a lot of time and resources. So being sure to make the right architectural decision is a very important question in a modern world of IT-industry.

Maintainability of a software system describes how well the system can be understood, repaired and enhanced. Maintenance is the main part of development process and occupies more resources and time than any other parts of this process. This means that one of the requirements on good architecture is making a project highly maintainable. Better maintainability is often the main criteria for architecture selection.

High performance of a software system or application is also important in many projects. Selecting the right architectural approach greatly contributes to achieving desired performance. Understanding of pros and cons of different

architectural approaches should be based on empirical researches, but there are not much research held to compare different architectures of Android applications, specifically VIPER, as it is quite new.

Android applications are of particular interest because of the relatively new technology and the fast growth [5]. Every software project is built upon an architecture. To understand the way to pick right architecture for a project with specific requirements and restrictions, the following questions will be answered:

- What architecture should be used to develop an application taking into account possible limitations, i.e., time, cost, desired performance?
- What architecture should be used to develop an application of specific size, i.e., small prototype application, medium size, big application with a big plan for extension?
- Which architecture follows SOLID principles better?
- Which architecture results in cleaner code?
- Which architecture suits better for modular application?
- Which architecture suits better for libraries?

There are few types of Android projects defined by the number of use cases added in the comparison. Use case is a set of actions that defines the interaction between a user and a system to achieve a goal. Small project is a project that has 0-6 different use cases. Medium project has 7-15 use cases. Big project has over 15 use cases.

Library is a type of project that solves one or two problems and is used by other software projects.

Modular project is a type of project that is usually big and built as a set of smaller projects that can function on their own.

RQ1	Which Android architecture is more maintainable, MVP or VIPER?
RQ2	Which Android architecture leads to better performance, MVP or VIPER?
RQ3	Which Android architecture fits better for different types of projects, MVP or VIPER?

1.4 Method

Literature review of different comparison papers [9][12] shows that when comparing different architectures, they rely mostly on one basic property – maintainability. To make such comparison, they use scenario-based methods. Clements, Kazman, and Klein offer a guide on how to evaluate a software architecture based on the following methods [6]:

- ATAM: Architecture Tradeoff Analysis Method
- SAAM: Software Architecture Analysis Method
- ARID: Active Reviews of Intermediate Designs

These methods evaluate properties of architectures with goals that are business related. One of these goals is cost reduction, for example. There are no direct business values for this thesis. Hence, cost reduction cannot be evaluated directly. The same restrictions are described in other scientific papers that evaluate architectures [14].

A method that will be used is similar to a method actually used to compare algorithms. A set of use cases will first be implemented using both architectures.

Then a set of metrics will be selected. These metrics include modifyability, testability, memory consumption and time, required to execute a use case. Then all metrics for architectures will be assessed and metrics data will be collected using the same scenario and setup.

A set of use cases is developed to evaluate architectures. Each use case consists of a few typical mobile app actions, such as switching screens, typing, accessing the database etc. Each use case is implemented using both architectural approaches MVP and VIPER. Then tests are executed on each use case, and metrics data are collected. For each use case, the application logic remains the same, the test execution environment is constant. The only difference is the architecture.

1.5 Contributions

Collecting metrics data and analyzing the results will answer questions, mentioned in the problem section 1.3. The metrics are performance, testability, and modifiability of an architecture, implemented in simple use case scenarios. The defined metrics and the collected data will be sufficient to define advantages and disadvantages of each architecture. The result of this thesis also include:

- a description of the two architectural approaches to building Android native applications,
- a definition of ways to compare architectures including a definition of suitable metrics,
- a collection metrics data for ten different Android use case scenarios implemented using two different architectures,
- a comparison of the two architectures based on the collected data and
- suggestions for selecting these architectural approaches for different tasks.

The thesis results create empirical data regarding the comparison of VIPER android architecture and MVP architecture. Related work does not contain any data describing the selection of the right architecture for different types of projects.

1.6 Target groups

Professional software developers, especially, mobile application developers, are the target group of this research. They will benefit from knowing answers to key research questions of this thesis. Being able to rely on empirical data might be crucial when making decision about architecture to select for a new mobile application. This decision will affect the costs of developing android projects project and their maintenance, as well as, the time required to finish development.

The Thesis describes the method to compare mobile architectures, so researchers are the second target group. Comparing mobile architectures is not a trivial task. The thesis defines key properties for architectures and explains these properties by means of representing those as a set of specific metrics that can be measured. The Thesis also describes how to interpret the results of this measurements. Future research in this field may use methods, described in this thesis, to compare other mobile architectures or to create other methods for architecture comparison.

1.7 Report Structure

The Background chapter 2 describes the basics of the problem more in detail. It explains what is interesting about this problem, why it should be investigated and how the solution will be used in practice. The Background chapter also describes previous research in this field and results obtained it that research.

The Method chapter 3 describes the scientific approach that is used to answer the research problem. It contains an explanation of the formal research methods used to answer research questions. It describes the experiment execution environment, the experiment setup and the metrics that are collected. The method is divided in steps that are explained in this chapter. Then there is an explanation of reliability and validity of the method used.

The Implementation chapter 4 contains details of the implementation of experiment. It contains a list of external frameworks that were integrated to conduct the experiment. The Evaluation chapter 5 shows and describes the results of the experiment. It shows the results of every step of the experiment, the analysis and the conclusions made. The Conclusions and Future Work chapter 6 contains overall conclusion on the topic of thesis and possible ways for future improvements.

2 Background

Bob Martin introduces a way to design a software architecture using layers[8]. These layers divide functions of the software system by responsibilities and dependencies. There are three main layers.

The first one is the Data Layer. This layer consists of classes and contracts with the main purpose of executing storage functions. These classes are responsible for writing, reading, updating and deleting data from local or external sources. These sources may include databases, external services, file system. Implementation of these layer depends on the operating system and protocols selected for this system or external services.

The second is the Business Logic layer. This layer consists of classes and contracts that describe algorithms of the main functions of the system. These components are responsible for executing business logic. This includes such operations as login, search, filtering, calculating different metrics. Components in the business logic layer depend on the contracts that are distributed by the data layer. These classes use components of the data layer to operate on data. Implementation of components on this layer should not depend on the operating system or any other external property.

The third layer is the Presentation layer. Classes and contracts of this layer are responsible for implementing interfaces to interact with the user. The user here is human or external service. In case of human, the user interface is implemented by means of the command line or graphical components. If the user is an external service, software system describes API. The implementation of the component on the presentation layer is split into two groups. The first group of components is responsible for the actual displaying of graphical elements. This group depends on the environment, that runs the software. The other group is responsible for executing presentation logic and handling view events. These events are clicking, scrolling, swiping. This group is independent of external properties. It uses the business logic layer to call corresponding functions.

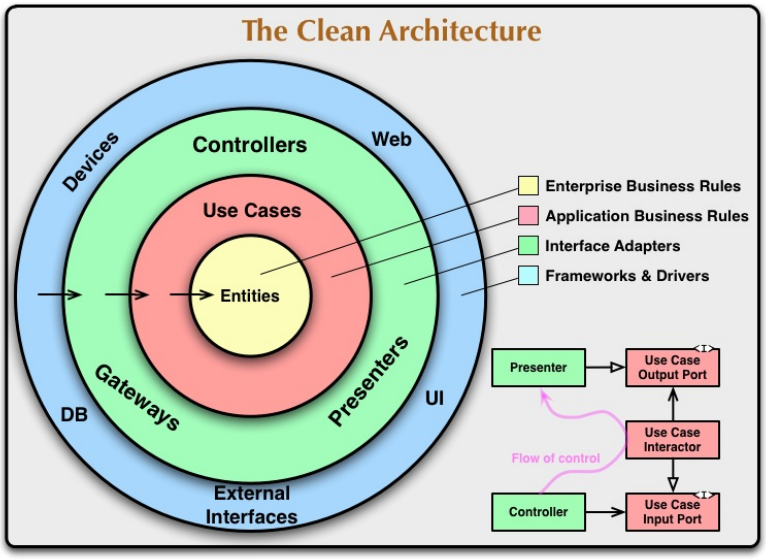


Figure 2.1: Architecture layers.

Both architectures follow base principles. These include SOLID principles. SOLID is an abbreviation that consists of the first letters of the names of the principles. These principles are formulated to establish ground rules for good system design.

P1. Single responsibility principle. This rule states that every component should have only one reason to be modified. This means that it has only one responsibility and the only reason for the component to be modified is the need to change corresponding responsibility.

P2. Open/Closed principle. This rule states that every component should be open to extension and closed to modification. This means that additional functionality should be implemented by means of an extension of existing components rather than modifying those.

P3. Liskov Substitution principle. This rule states that entities in the software system should be able to be replaced with their subtypes without affecting the correctness of the system. This means that if an entity has a subtype, this subtype should implement all the contracts of the base component.

P4. Interface segregation principle. This rule states that component contracts should be minimal and describe only one set of rules. Components in software system operating on a set of interfaces that are called contracts. These contracts specify responsibilities of the component without relating to the specific implementation. This rule introduces a way to structure contracts. Contracts should be minimal. The behavior of the component should be described by a composition of small contracts rather than by one big contract.

P5. Dependency Inversion principle. This rule states that component should depend on abstractions and not specific implementations. This principle is easily implemented by means of interfaces and contracts.

These principles aim to set some ground rules for architecture design. They define contracts as the main way of the communication between components. This leads to more abstraction in the implementation of the software system. A lot of abstractions means that new components are easier to integrate into the system, as the only requirement is for a new component to implement the contract.

Modifying a component in a big software system is a tricky task as there are other components that depend on this one. Using contracts mean that no component depends on the actual implementation of the other component, but it depends on the contract. This means that modifying one component will not affect system correctness if it implements the contract.

2.1 Android platform

The Android operating system is built upon a few major components and layers. The top layer is the application layer. All applications are executed on this level. To access operating system functions, application use API for Application Framework layer. This layer also contains native applications. These applications are provided by the implementation of the operating system. Web-browser and email application are the examples of native applications.

The application framework is responsible for providing device and system data via API to applications. It collects data from different sensors and hardware. This layer builds the environment for the applications to run. It uses different android services for this purpose. Some of these services are:

- activity manager (contains activity stack);

- resource manager (provides access to the resources of the application, like images and strings);
- locations manager (allows applications to get updates on the location data);
- notification manager (provides application the means to display notifications);
- view system (creates view components).

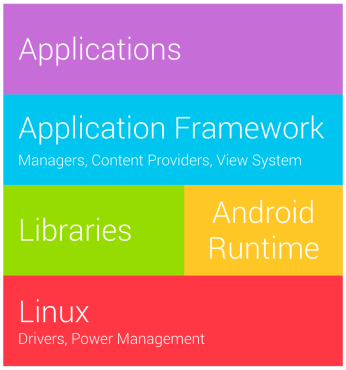


Figure 2.2: Android operating system architecture.

For this purpose, it uses system libraries and Android Runtime environment. Android system libraries and Android Runtime environment are on the same layer and are responsible for providing basic functions and procedure calls. Android libraries give access to all the basic functions. Some of the libraries:

- android.text (used to display text on the screen);
- android.media (used to play video and audio);
- android.database (provides access to the database);
- android.hardware (used to access hardware sensors).

The most basic layer is the Linux kernel. It is the core of the operating system, it contains drivers to hardware components and operates with those. The kernel handles primitive multitasking, power management, process management.

The android application consists of base components. These component are Activity, Fragment, Service, Broadcast Receiver, and Content Providers. Each component has its own lifecycle.

Activity is the main component in building an android application. It is the only component that has a graphical interface. Fragments are the part of the activity and also have a graphical interface. Activities usually represent a single screen of the application. Fragments are used to enhance the user interface inside the activity.

Android application is a set of activities, that change each other. This should be taken into account when designing the Android architecture. Activity has a specific lifecycle and different actions and calls should be executed on different steps of this lifecycle.

Method onCreate is called when the activity is created. Main components should be defined and initialized on this step. Method onResume is called when the user will be able to start the interaction with the graphical interface. Method

onPause is called when activity is no longer visible. Method onDestroy is called right before the activity is destroyed.

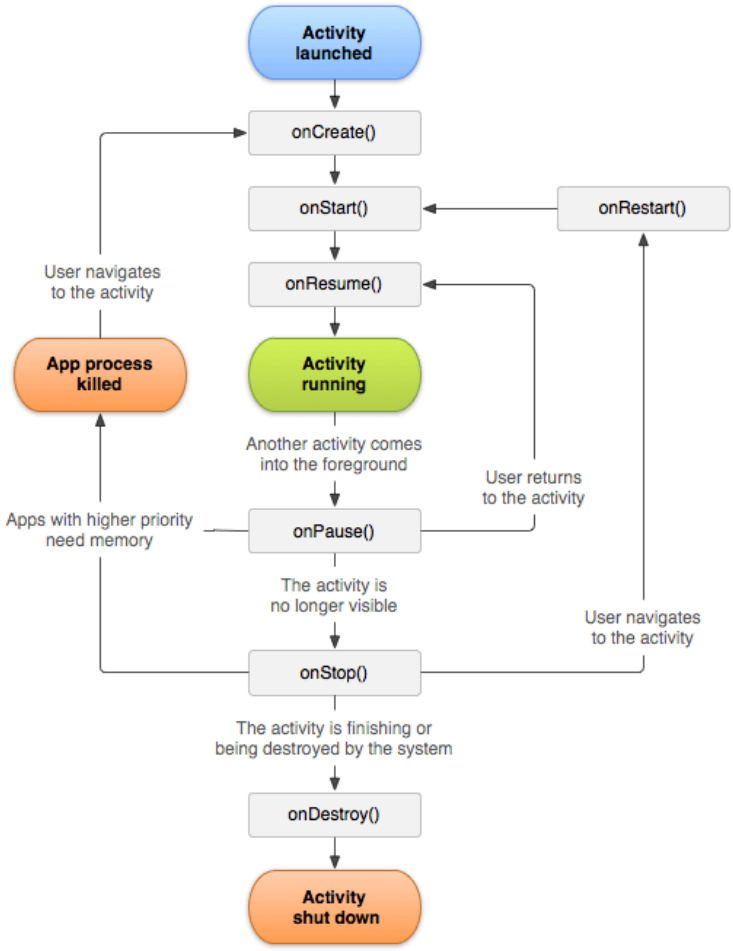


Figure 2.3: Activity lifecycle.

Android architectures should take this into account as the activity is the only instrument to operate with the graphical interface and using it incorrectly will cause errors. For example, making a call to an activity, that is already destroyed, will cause a crash of the application. Architecture components should always be in sync with the Android classes state.

2.2 Existing architectures

There are three most popular mobile architectures used in Android projects in modern application development. These architectures are MVC (Model-View-Controller), MVP (Model-View-Presenter) and MVVM (Model-View-ViewModel).

MVC is short for Model-View-Controller. This architecture describes three main components: model, view and controller. Model is responsible for the definition of the data structure and it updates the application to reflect the current state of data. The controller in this architecture contains control logic. It is responsible for receiving updates from view and notifying model about these

updates. The view is responsible for displaying the user interface. It handles all user inputs and sends it to the controller. The structure of this architecture is displayed in figure 2.4.

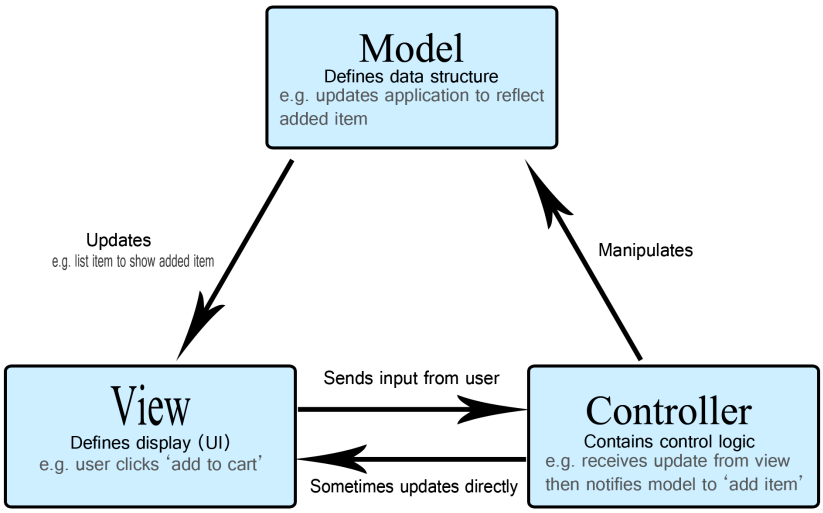


Figure 2.4: MVC structure.

Main feature of Model-View-Controller architecture is that Model knows and directly linked to View to be able to change it according to the current state of data. The controller is lightweight and only notifies model about data changes that are triggered by user input. View in this architecture handles a lot of responsibilities as it should handle all the user input, navigate those to the presenter, and handle all the updates from the model and presenter.

MVC architecture was a concept that enabled parallel development, as different parts of the architecture could be developed by different people simultaneously. It also was a way to make software system components reusable. The view can be changed without any changes in the model or controller. The same is true for controller and model. However, a direct link between the model and view creates a loop of dependencies, which means that the same model is hard to use with different views. It requires the view to either implement some abstract contract or create different models for different views. New architectures were designed to fix this issue.

MVVM is short for Model-View-ViewModel. The model represents the data access layer. It describes the data structure of the software system. It receives updates from the ViewModel component and updates the state of this component. The view is responsible for displaying the user interface. It also handles user input and directs it to the ViewModel component. ViewModel is the component that is tightly connected to the view. It stores the current state of the view and data. It is responsible for updating the view, handling user input received from the view and updating data in response to this input. The structure of this architecture is displayed in figure 2.5.

This architecture requires a specific binding technology to function correctly. Binding is required to directly connect the data state stored in the view model to

the view. XAML (Extensible Application Markup Language) is the binding technology implemented by Microsoft to initialize structured values and objects. It is a part of .Net Framework and is used in WPF (Windows Presentation Foundation). This framework is mostly used to develop Windows store applications and mobile applications for Windows Phone.

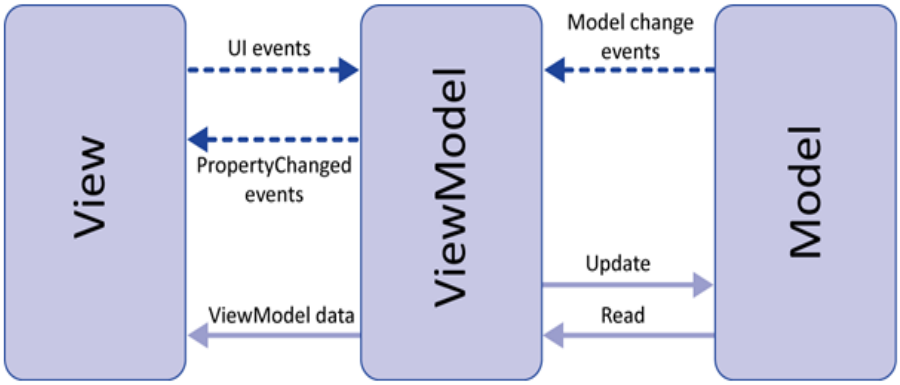


Figure 2.5: MVVM structure.

Google developed its own framework that implements the binding technology required for MVVM architecture. The Data Binding Library is described on the official Android developers website. This library is relatively new and still unexplored as well as MVVM approach in Android software development. This architecture fully depends on one specific framework not just for convenience, but for being able to function.

MVP is short for Model-View-Presenter. This architecture is the most used in the Android application development now. This approach allowed to keep the advantages of MVC like the ability for parallel development and fixed its issues. The model is MVP is not connected to the View and it became possible to reuse the model without any restrictions. It is particularly good because the model represents the mechanism that provides data in the system. This means that the independent model can be used separately. For example, it is possible to implement an Android background service that has no user interface for the application, using the same model, that is used in the application activities with a graphic interface.

2.3 MVP architecture

MVP is more traditional architecture [16]. MVP is Model-View-Presenter. The model here is responsible for storing domain objects and performing logical operations on them.

The model usually refers to the Business Logic layer. The business logic layer provides contracts of components that retrieve the software system data and execute application logic on it. Data access layer is also a part of the model as it is directly responsible for executing main storage operations like create, read, update, delete. The view is responsible for building the user interface. It contains the logic to initialize every user interface component in the right place of the screen and handle user input events. These user interface components are buttons, labels, scrollers, dropdowns. And user input events are clicks, scrolls, swipes. The presenter is responsible for handling all callbacks from view and uses the model to get data, required for the view to build the right user interface. It is the mediator

between view and model. The presenter receives all user input events from the view and makes calls to the model. It also receives responses from the model and updates the view.

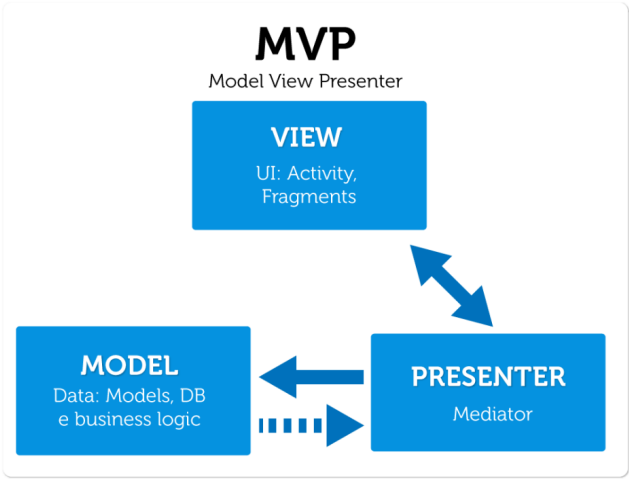


Figure 2.6: MVP architecture.

One of the big advantages of Model-View-Presenter architecture is that it describes three levels of abstraction. This makes it easier to debug the android application. The separation of business logic and data access logic from the view also enables unit testing to be implemented.

Model-View-Presenter architecture in Android provides a better separation of concerns. It moves business logic and data access logic out of Activities and Fragments. Activities and Fragments are platform specific classes, and business logic should be platform independent. Model-View-Presenter software architecture makes the application more reusable. Presenter, view, and model are connected via contracts, which makes them independent of the specific implementations. Every presenter can easily be replaced by other presenters, that implement different logic if it implements the same contract.

2.4 VIPER architecture

VIPER architecture is relatively new. VIPER stands for View-Interactor-Presenter-Entity-Router. The view is responsible for showing user different items of the user interface, like progress bars, buttons or dialogs. The view also handles user input events. User input events are then passed to the corresponding methods of the presenter.

Presenter handles all callbacks from view and uses interactor to get data, required for the view to build the right user interface. It also executes all the presentation logic, for example, makes decisions to show or hide progress bar, or show a dialog. The interactor is responsible for performing any logical operations on domain objects. Interactor represents one logical operation, for example, search or filtering. The presenter usually uses few interactors. The interactor uses entity components to get domain objects and perform business logic operations on that data. The entity is responsible for storing domain objects. It provides the interface to access data storage and execute basic operations like create, read, update and delete. The router is responsible for navigation between different scenes of the user interface. The router is accessed by the presenter.

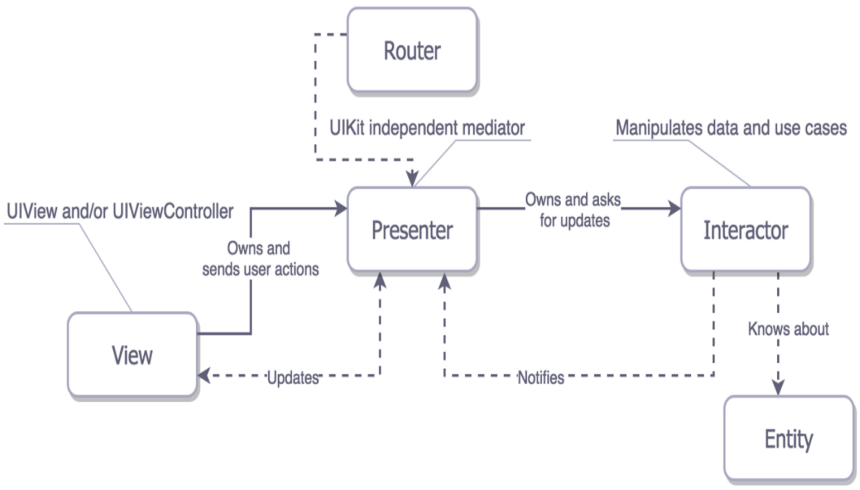


Figure 2.7: VIPER architecture.

The great advantage of View-Interactor-Presenter-Entity-Router software architecture is that it provides good testability out-of-box. The main feature of the VIPER software architecture is a loose coupling between components. Every component is designed to have the smallest range of responsibilities and perform the least set of actions. This comes in hand when implementing unit tests for the project. Every component is easy to test, as all dependencies can be mocked. View-Interactor-Presenter-Entity-Router architecture is originally an attempt to apply SOLID principles on iOS platform. Migrating to the Android platform, the architecture preserved the orientation on following the basic architecture design principles.

The VIPER software architecture features independent modules, which makes it easy to work on the application for a big team of developers. The basic skeleton of the architecture is implemented first, and all specific modules are then assigned to different developers to implement. All the implementations of the components are based on the same skeleton, so the project looks solid.

The codebase of the project using the VIPER architecture looks very similar. This makes it easy to understand the project for new members of the team, that are familiar with the VIPER architecture. The code is not hard to understand for a person, that is familiar with VIPER, but is not familiar with the specific platform. For example, android developer is able to understand and iOS project, because components are similar and the structure is the same.

2.5 Architecture comparison

Both these architectures, MVP and VIPER, follow base architecture principles. There are few differences in components structure and responsibilities, which results into different implementation of project and difference in such properties as modifiability and testability.

MVP and VIPER have similar components. But they have different responsibilities. Presenter in MVP is responsible for handling callbacks from view, executing business logic and receiving data from the model. VIPER architecture defines a separate component, interactor, for executing business logic,

so the presenter is more simple. View in MVP is responsible for building the user interface, as well as navigation between different views. VIPER introduces separate component, router, that handles all of the navigation. It is not connected to view at all, it operates by presenter command.

2.6 Android application use case scenarios

The android application use case scenario is a set of basic action that needs to be performed to complete one use case. These actions are described from the point of view of the application. Basic actions are the following: click the button, access the data storage, open a new screen, type text, select item from list, switch tabs etc.

Most popular Android applications usually have a small or medium size of use case scenarios. Small are use case scenarios that consist of up to five basic actions. Medium use case scenarios consist of five to ten basic actions. That can be viewed in the most popular Android applications.

Twitter is one of the most popular application for Android. This application allows users to post short messages. The most popular use case scenario here is to post a tweet. It consists of following basic actions:

- click a button to start tweeting;
- open new tweet screen;
- type message;
- click the tweet button;
- post tweet on the backend (access data storage).

Another popular use case scenario in the Twitter application is to read the user feed. Basic actions here are the following:

- open user feed screen;
- load tweets in use feed (access the data storage);
- scroll to view tweets.

These scenarios are the most popular use case scenarios in the application. One of these scenarios has three basic actions, the other has five. Both these use case scenarios are considered small, because of the number of actions.

Another example of a popular Android application is Netflix. This application allows a user to watch different TV-shows online. The most popular use case scenario here is to open the latest episode of the TV-show, that user watches regularly. This scenario consists of the following basic actions:

- load recently watched TV-shows (access the data storage);
- click on the TV-show icon;
- open new screen to stream the video;
- load the latest episode (access the data storage).

Another popular use case scenario is to search for a TV-show and start watching it. The basic actions list here is the following:

- open search screen;
- type the title of the TV-show;
- load the list of TV-shows, matching the search query (access the data storage);
- click on the TV-show icon;
- open new screen to stream the TV-show;
- load the TV-show (access the data storage).

These use case scenarios are the most used in the Netflix application as they correspond to its main idea. These scenarios consist of four and six basic actions. This number of actions correspond to small and medium size of use case scenarios.

A calendar is the application, that is installed on every mobile device. Main use case scenarios here are viewing events for current day/week and creating a new event. Viewing events for a week is a use case scenario that consists of the following basic actions:

- show the screen with a list of tasks;
- load all the events for a current week (access the data storage);
- scroll to view all the events.

The other popular use case scenario is to create a new event in the calendar. It has the following set of basic actions:

- click the create button;
- open new event creation screen;
- select the date of the event;
- select the time of the event;
- type the name of the event;
- type the description of the event;
- click "save" button;
- create a new event in the database (access the data storage).

The first use case scenario has only three basic actions, and the second consists of eight. These use case scenarios describe the main purpose of the application. One of the calendar main use case scenarios is small, and one is medium size.

The other mobile application present on any device is email box. This application gives a user the ability to view and send emails. And these are the most popular use case scenarios for this application. To view the email, the following steps should be completed:

- open email list screen;
- load all emails (access the data storage);
- select one email;
- open the new screen to view the email;
- load selected email (access the data storage).

The use case scenario of writing a new email consists of basic steps listed below:

- click "new email" button;
- open a new screen to compose an email;
- type the recipient address;
- type the title;
- type the body of the email;
- click "send" button.

Both these use case scenarios have the number of basic actions not greater than six. Application that enables a user to send and receive emails is the most basic one for every mobile device. Listed use case scenarios don’t contain any complex computations.

Youtube is one of the most popular and most used applications for Android mobile devices. The most popular use case scenario in this application is to view a video. To find a view and view it, the following basic actions should be completed:

- click the "search" button;

- open search screen;
- type the search query;
- load the list of video according to the search query (access the data source);
- click on the icon of the video;
- open new screen to stream a video;
- load a video (access the data source).

This data access scenario, that includes search, consists of seven basic actions from the perspective of the application. These use case scenarios contain networking, but no complex computations or excessive multitasking. Most popular use case scenarios mostly have a goal to enable a user to achieve goals faster.

Example of these five popular Android applications with hundreds of millions of downloads shows that the most popular use case scenarios consist of a small number of basic actions. This means that use case scenarios of a small or medium size are of particular interest when evaluating application architectures.

2.7 Literature review

Existing research on this topic mostly consists of different thesis describing comparison of other android architectures or methods to evaluate android architectures.

Tian Lou made research in this area in his work "A Comparison of Android Native App Architecture– MVC, MVP, and MVVM" [1]. Key metrics were selected to compare these architectures. These metrics are performance, modifiability, and testability. Collected data was later compared and analyzed to define the better architectural approach.

Analysis and experiments show MVP and MVVM have better testability, modifiability (low coupling level) and performance (consuming less memory). That is, MVP and MVVM are better than MVC on the selected three criteria. But for MVP and MVVM, there is no evidence showing that one is superior to another. These two architectures have similar performance, while MVP provides better modifiability and MVVM provides better testability [1].

“Structuring Android Applications” by Anthony Madhvani[2] describes ways to combine different architectural approaches (MVP, MVVM, MVC) with different design patterns as well as frameworks to build the Android application. This work also describes which architecture to select for different projects according to the specifics of the project.

Hugo Kallstrom in his work named "Increasing Maintainability for Android Applications" compares different architectures (MVP, MVC, and MVVM) to define which is more maintainable, modifiable and testable [3].

Analysis of Android vulnerabilities and modern exploitation techniques [4] describes importance of developing applications with good architecture to make it less possible to be exploited. Key Architecture Principles:

- Build to Change Instead of Building to Last
- Separation of Concerns
- Single Responsibility Principle
- Identify Components and Group them in Logical Layers
- Do not Repeat the Functionality

These principles mostly require architecture to be maintainable. Building to change instead of building to last means that application project will change a lot and it is important to design it in a way to reduce cost of changing it. Following principles describe how to do that. Overlapping of components functionalities, multiple responsibilities of components, repeating of functionality will make it harder to understand project and change it.

2.8 Summary

The background chapter covers the basic knowledge about software architecture, it's properties and motivation. The software architectures are designed to structure the software project is a specific way. This is important to help software development companies to better achieve their goals. Software development companies aim to deliver a software system in the shortest terms with the lowest expenses and highest quality. These requirements developed into the definition of the most important properties of the software architecture. These are maintainability and performance. Maintainability is described as a composition of modifiability and testability. This chapter also explains the two architectures, that are compared in this thesis, Model-View-Presenter and View-Interactor-Presenter-Entity-Router. These architectures have similar components but the responsibilities of these components differ.

A literature review describes thesis and articles regarding the comparison of software architectures and the properties of architectures. Architectures are compared based on their properties. Main architecture properties are first evaluated on simple and basic examples of applications. These properties are then compared and analyzed. Architecture properties define whether the architecture is suitable for one type of the software project or another. For example, startups usually require the first version of the software system to be developed as fast as possible. This means that such properties as performance or testability may be sacrificed for higher modifiability.

3 Method

Architecture comparison is an empirical research task. Methods used to compare architectures are based on experimental research. Experimental research is conducted in a form of a set of tests. To do this, a stable environment is created, test cases are designed. The experiment has a purpose of proving a scientific theory with defined dependent and independent variables. Experiment environment and variables are described in section 3.1, experiment body is described in section 3.2.

The method that is used is similar to the method used to compare algorithms. Example apps following the architectures will first be implemented, set of metrics will be selected. Then all metrics for architectures will be collected using the same scenario and setup. A set of use cases is developed to evaluate architectures.

Each use case consists of a few actions, such as switching screens, typing, accessing the database etc. Each use case is implemented using both architectural approaches MVP and VIPER, and then tests are executed on each use case, and metrics are collected. Types of use case scenarios and basic actions, that they consist of, is explained in the Background chapter 2.6 Android application use case scenarios. It also motivates the size and complexity of test use case scenarios.

This research measures such properties as performance, testability, and modifiability. The main property of an architecture of a project is maintainability. Testability and modifiability contribute towards this property, so these will be measured. Performance should be measured because architectural requirements state that there should be proper memory management and user interface should not be affected by any lags or delays.

3.1 Scientific Approach

The research starts with the Literature Review. This method is used to establish the background of the research and select the main method and means to implement the experiment that helps to answer research questions. The result of systematic literature review in the field of software architecture design and mobile architecture comparison is specifically defined main architecture properties and general approach to compare architectures.

Based on the literature review, the experiment is designed to evaluate Android software architectures. Software architectures, MVP and VIPER, are evaluated by measuring a set of properties for both architectures in a stable environment. Data, obtained in the experiment, is analyzed to compare architectures and answer research questions. Direct comparison of the properties of architectures is enough to answer research questions RQ1 and RQ2. Additional analysis is required to answer the research question RQ3. The measurements, obtained from the experiment, are sufficient to answer this question.

3.2 Method Description

Numerical data needs to be collected for all metrics for both architectures, so controlled experiment method is used. Stable and constant execution environment is selected to conduct experiment. This is the physical Android device, table 3.1 contains device technical specification.

Device name	LG Nexus 5X
Operating system	Android 8.1.0
Chipset	Qualcomm MSM8992 Snapdragon 808
CPU	Hexa-core (4x1.4 GHz Cortex-A53 & 2x1.8 GHz Cortex-A57)
GPU	Adreno 418
RAM	2Gb
Resolution	1080 x 1920 pixels, 16:9

Table 3.1: Device technical specification.

Independent variable in this experiment is which use case is currently running. Dependent variables are performance metrics, modifiability and testability. The way how these metrics are measured described in 5.1 Experiment description. Performance metrics are memory consumption and time required to execute a use case scenario. All user actions are performed by an automated testing framework, so they always take constant time. This means that clicking the button or scrolling the list always takes the same amount of time. Automation here is required to make sure, that the only factor affecting performance metrics measurements is the software architecture design.

Modifiability metrics are the number of classes that need to be added to implement a feature, number of classes that need to be edited to implement a feature, and a number of lines of code, that need to be added or modified to implement a feature. These metrics describe amount of mechanical work, required to implement a feature. Implementations of all use cases using both architectures rely on the same frameworks. This means that the amount of code depends only on the actual architecture design and number of components that this architecture needs to run.

Testability is described by the test coverage metric. Test coverage shows what part of the total amount of classes in the project is covered with tests, i.e. executed in the test environment to validate the correctness. The most common approaches in the software development industry were selected to implement test cases. Unit tests are used to test every single component. Integration tests are implemented to verify the correctness of the work of whole use cases, that requires few components to work together. User interface tests ensure that graphics components are displayed correctly and respond to user actions. Numerical data collected in this experiment is later used to compare and analyze architectures to provide qualitative results.

Few types of projects are included in comparison to define what architecture would suit these types of software projects better. Each type of project has different weights for every metric collected in the experiment.

The most important metric for small and medium project is number of classes that need to be added to implement a feature. Small projects are mostly prototypes that need to be implemented in the shortest terms possible. Performance and testability metrics are not important as project will most probably not contain any complex business logic.

Testability and performance are important metrics for big projects. Testability makes it easier to implement new features and being sure to not affect the correctness of the implementation of previous features. Number of classes that

need to be edited to implement new feature is important as well, as big projects usually contain a lot of classes and changing those may affect the correctness of the system.

Software libraries value such metrics as number of classes that need to be added to implement a feature and number of lines of code. Libraries are used as parts to implement other projects and need to have small resulting size, as projects usually use few libraries and resulting application may be quite heavy.

3.3 Reliability and Validity

The experiment aims to compare architectures by performance of use cases, constructed using these architectures. Performance is measured in terms of time of execution and memory used. Collected results will not have absolute values, but will be valuable relative one to another. Business logic will be the same, so it will not have an effect on the results of the comparison and the only different thing in applications will be the way of code organization structure. Architectures describe components and relations between components, so these will not depend on the person, who implements the architecture. Application logic, presentation logic and data access logic stays the same for test use case scenarios, so these factor will not effect the results of comparison. This means that architectures may be compared and way of implementation of each concrete class will not have any effect on results of the comparison.

The main goal of this method is to collect data on the same metrics and using the same setup for different architectures. This means that measurements are valid to compare metrics for architectures, but will not present absolute values for architecture benchmark.

Typing and clicking controls speed is constant, as these are performed by automated framework. Drawing and loading speed is constant, as device setup is constant. Only factor affecting amount of memory required for application and time is amount of operations that needed to be loaded, which is different for application architectures.

Results may not be representative for big projects with very complex business logic, as sample use cases are not big. Also, these results will not be valid for game applications, as these are developed using very different approach and have different goals.

Also this way of evaluating architecture testability may be misleading on its own. Tests may not be good and may be misleading. High test coverage as the main goal of a project architecture leads to focusing more on testing, but not on achieving the desired behavior. And implementing the right behavior of architectural components is the main goal of any architecture.

The experiment is executed on the small use case scenarios. These use case scenarios consist of a small to a medium number of simple steps. These steps are default actions that are executed during the work of the application. These actions are opening new screens, clicking buttons, typing text, loading data, executing algorithms, accessing the database and other. Use case scenarios presented in the experiment consist of opening two/three screens, accessing the database, loading data, saving data, clicking buttons and typing text.

Most of use case scenarios for a mobile application are not big. For example, there is a popular application called Instagram. It allows a user to post photos, view photos of his friends and like photos. Like a photo - this is one of use case scenarios for this application. From the point of view of the application it consists

of the following actions: get user feed from the backend (access the data storage), scroll the feed to the photo, press the like button, send a request to the backend (access the data storage). It consists of only four basic actions and doesn't need to change screens.

Another scenario is to post a photo. Actions here are the following: press the button to start posting, open a new screen, pick the photo, open a new screen, choose a filter, apply selected filter, open a new screen, type caption, click the "post" button, create the post on the backend (access the data storage). This use case scenario is longer, than the previous one, and consists of opening three new screens. Most applications consist of both short and medium use case scenarios. However, the developers of big applications tend to minimize the number of actions and screens required to complete a use case scenario as users tend to get lost or to lose the will to finish the scenario.

The experiment is conducted on small and medium use case scenarios. These scenarios are described in chapter 5.1. Results of the comparison may not be representative for any android application as the test scenarios have some limitations. Test scenarios consist of a small or medium amount of actions (up to ten). Test scenarios don't do a lot of parallel computation. It uses multithreading to execute business logic and access the data storage, but no complex computations.

The experiment is conducted on only ten different use case scenarios. These test scenarios represent the most common scenarios in mobile applications, as they contain basic actions. A higher number of test scenarios would increase the validity of the results. Implementing new test scenarios is a subject for future research.

The thesis aims to compare two android application architectures implemented for the most basic applications and use case scenarios. These use case scenarios are the backbone of the application and then enhanced with more complex ones. Performance of an architecture in applications with features like executing complex computations, big use case scenarios or working with a lot of data can be investigated further in future work.

The reliability in performance metrics measurements is achieved by administering the same test few time. Memory consumption and time required to execute a use case scenario are measured on the same set of use case scenarios in the same test environment. One test consists of 500 executions. This test was conducted two times and measurements were compared to ensure that results don’t change over time.

To ensure that architecture comparison results could be reliable, all tests are executed and all metrics are collected on ten different use case scenarios. The same results of comparison for each use case scenario means that these results have internal consistency reliability.

All the use case scenarios are implemented by the same person, which is a flaw in reliability of the research. Metrics to evaluate architectures were designed to minimize the effect of this flaw on the final result. Obtained measurement for both architectures are still valuable in comparison, but not as absolute values. Inviting more software developers with expertise in Android development to implement the same use case scenarios would improve the reliability of the results.

4 Implementation

All use cases are implemented in form of android applications. These applications require minimum android API level 16, which corresponds to Android 4.1 Jelly Beans. 99% of the Android market operates on the Android version 4.1 or higher, according to official Android dashboards [20]. Android application is implemented in Java programming language. Unit tests are implemented in Java, using JUnit and Mockito testing frameworks. UI tests are implemented in Kotlin programming language, using Espresso test framework.

These frameworks are used in the implementation of test use case scenarios for convenience. The same set of Android frameworks used in the development of both VIPER and MVP use cases. These frameworks are used by Android applications developers in a huge amount of applications. Java is the most used programming language for developing Android applications. Kotlin is the programming language that is relatively new but officially supported by Google as a language for developing Android applications with most official tutorials containing examples in both Java and Kotlin.

Applications implementation is based on a few frameworks. Google Room is used tom implement data access layer by means of data access object pattern. Dagger 2 framework is used to implement dependency injection. RxJava and RxAndroid frameworks are used to follow principles of reactive programming. The selection of frameworks to work with database or implement dependency injection is not important. The main point is to follow the same approach and use the same frameworks when implementing test scenarios for both architectures.

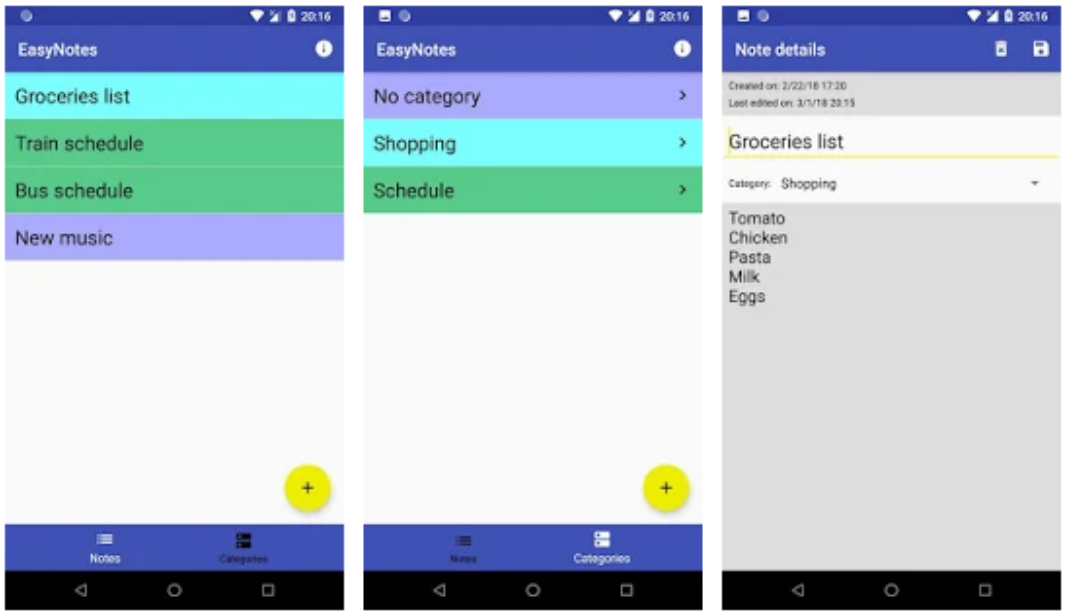


Figure 4.1: User interface.

Figure 4.1 shows user interface of a few screens in implemented use cases. User interface uses modern approach in design, developed by Google, called Material design. Navigation between main screens is implemented in form of bottom navigation bar. Floating action button is used to raise users attention to main action on the screen, such as creating new note or category. Menu is used to represent set of control actions for every screen, such as save or delete.

Test cases are split into categories. The first category is represented by unit tests. These tests are implemented using the Java programming language. The main purpose of unit tests is to ensure that every single component of the software system is working correctly. These components are classes in the data access layer, domain layer, and classes in the presentation layer, that execute view logic. To make sure that classes, which specific component depends on, don't affect the results of the test, these dependencies are mocked using special testing framework Mockito.

The second category is integration tests. These tests are implemented using the Java programming language. The main purpose of integration tests is to make sure, that components work correctly together. This means that these tests cover specific data flow and activity scenarios. As soon as every component is covered with unit tests, results of integration tests depend only on the correctness of the implementation of the specific scenario.

The third category is user interface tests. These tests are implemented using Kotlin programming language. The main purpose of user interface tests is to ensure that every graphics component is displayed at the right time and is responsive to user actions. These tests cover classes of the presentation layer, that are responsible for displaying graphics components. All three categories cover a different set of classes and create different test coverage. Jacoco testing framework is used to combine the test coverage report for the whole project.

Android use cases are implemented using Activities and Fragments. Activity is a basic Android user interface component. It is a component that displays graphics components on the screen and allows the user to operate on these components. Fragment usually provides a behavior or a part of the user interface to Activity. User interface components like buttons, text views and others are split between Activity and fragments. The bottom navigation bar is displayed by Activity because it stays on the screen all the time. Action bar (top bar) is also attached to the Activity, but options menu changes according to the active fragment. Other user interface components are attached to fragments. Separate fragments contain such graphical components as lists, text views, edit texts, images, and buttons.

Room framework is used to access the database. The room is a framework designed by Google to implement the Data Access Object pattern. This pattern suggests creating a separate object to execute create, read, update, delete operations for every entity in the database. Room framework also provides instruments to update the database version and implement data migration.

All operations are executed in the background thread. This is necessary to avoid freezes or lags in the user interface. The RxAndroid framework is used to implement the mechanism of asynchronous calls and synchronous execution of callbacks. This framework relies on the worker thread to execute all complex computations or operations and returns to the main application thread when work is done.

Dependency injection technique is widely used in the project implementation. This approach specifies an object that provides all other system entities and components with required dependencies. Dependency injection is a way of implementing the Inversion of Control principle. As the project uses a lot of abstractions and classes depend on contracts (interfaces) and not on specific implementations, objects are provided with all dependencies by an outside component. Objects get dependencies that fit contracts, without specifically knowing about the implementation of these contracts. Dagger 2 is the framework that handles dependency injection. Dagger processor receives a set of special

classes, called components, that provide implementations for the contracts that classes inside the system depend on. Then, dagger processor receives special interfaces, called modules, that describe the connection of dagger components with classes in the software system. Dagger processor generates injector classes automatically to provide software system with all required dependencies on the compilation time.

The graphical user interface is implemented using XML layouts. One XML file contains a description of graphics components that will be displayed on the screen. It contains all properties and specifies identifiers for all the components so that they can be later addressed from the code. Butterknife is the framework that automatically, using Java-annotations, binds Java objects with XML descriptions of graphics components.

5 Experiment description and results evaluation

The experiment description and results evaluation chapter contains the detailed description of the implementation of the experiment. The experiment implementation consisted of the set of use cases, that execute basic operations in the Android application. Every use case was designed to represent different types of use case scenarios. These scenarios consist of a different number of simple actions. Some actions are logic heavy and require more time to be performed, other are simple and lightweight. Some actions are pure hardware calculations, other are user input actions. This chapter also contains the tables with the measurements, obtained in the experiment. This data is analyzed to answer the research questions. Architectures are evaluated in the context of the measurements and compared to answer research questions.

Software architectures are evaluated as a composition of three properties. These properties are modifiability, testability, and performance. Modifiability metrics are the number of classes that need to be added to implement a feature, number of classes that need to be edited to implement a feature, and a number of lines of code, that need to be added or modified to implement a feature. Testability is described by the test coverage metric. Performance metrics are memory consumption and time, required to execute a use case scenario.

Performance is important metric to evaluate software architecture. Bad performance leads to the user having a bad experience of using the piece of software due to possible lags and freezes in the user interface. A bad performance will also lead to the user experiencing a long waiting time when the application opens new screens or loads data. Waiting for a response after performing a user input is also a consequence of a bad performance. These actions are considered as basic and frequent in the work of a software system. So performance should be one of the priorities in designing a good software architecture.

Modifiability represents the ability of a project to be updated. It includes both implementing new features and fixing bugs. Good modifiability will lead to less time and resources spent on adding new features. Fixing bugs in the existing code of the project is inevitable and good modifiability makes it easier to find the source of errors.

Testability is evaluated by means of test coverage metric. This metric shows how many components of the system may be covered with tests. There are a few types of tests: unit tests, integration tests, and user interface tests. All these tests are implemented and combined test coverage report is the basis for the evaluation of testability.

5.1 Experiment description

Use cases to evaluate architectures are implemented in domain of a simple notes application. There are possibilities to create, edit and delete notes, create, edit and delete note categories and add notes to categories.

Use cases are following (brackets contain list of actions):

- create note (open create note screen, type title, description, select category, click “save” button, create note in database);
- edit note (read all notes from database, open edit note screen, load note from database, change title, change description, click “save” button, update note in database);

- create category (select “categories” tab, open create category screen, type title, select color, click “save”, create category in database);
- edit category (read all categories from database, open edit category screen, load category from database, change title, change color, click “save” button, update category in database);
- delete note (read all notes from database, open edit note screen, click ”delete” button, confirm deletion in dialog, delete note from database);
- delete category (open categories tab, read all categories from database, open edit category screen, click “delete” button, confirm deletion in dialog, delete category from database);
- create category note (open categories tab, read all categories from database, select category, open create note screen, type title, description, click “save” button, create note in database);
- edit category note (open categories tab, read all categories from database, select category, read all notes from database, open edit note screen, load note from database, change title, change description, click “save” button, update note in database);
- delete category note (open categories tab, read all categories from database, select category, read all notes from database, open edit note screen, click ”delete” button, confirm deletion in dialog, delete note from database);
- show info dialog (click “info” button, open information dialog).

All these use cases consist of smaller steps. These steps are common for different use cases, and different use cases are different combinations of these actions. Actions are either user interaction or access to application infrastructure. Metrics are collected for each of these 10 use cases. Metrics are performance, modifiability and testability.

The ISO/IEC 25010 standard presents a software quality model where performance is grouped into three categories; capacity, time behavior and resource utilization [7]. The performance will be measured using the Android Profiler developed by Google. It was developed to measure different areas of performance for an Android app such as CPU, GPU, memory, battery or network. This data can be monitored during the process of its collection.

The performance is calculated as a set of two measurements, collected at the same time during the execution of a use case. These will be response time and memory consumption. A similar set of actions is performed on each use case for both architectures. All sets of actions will correspond to different user actions. Each set is performed and metrics are collected for both architectures 1000 times to reduce error.

Performance tests are implemented using Espresso testing framework. Typing and clicking controls speed is constant, as these were performed by automated framework. Drawing and loading speed is constant, as device setup is constant. Only factor affecting time is amount of operations that needed to be loaded, which is different for application architectures.

From the ISO/IEC 25010 software quality model, modifiability is defined as “Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality” [7]. Modifiability is described by 3 main measurements. To evaluate modifiability of the project architecture following questions regarding adding new feature should be answered:

- What classes need to be modified and how many?

- What new classes need to be added and how many?
- How many new lines of code (LOC) need to be added or deleted?

These metrics will be collected for each feature in use cases with MVP and VIPER architectures. Use cases are split into small tasks, so they are atomic and can represent different aspects of mobile application development and create a high number of collected values to understand the global tendency better.

ISO/IEC 25010 defines testability as “Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met” [7]. Few years ago, there were not enough tools to implement good tests for mobile application project [11]. And this was a real challenge, as unit testing was already one of the most important parts of the process of quality assurance. But in recent years lots of open source tools for testing became available, such as Robotium [18] or Mockito [19].

One way to measure testability of code structure is a metric called code test coverage. It shows which classes are covered by tests, which methods and parts of the code are testable. These measurements are aggregated for the whole project and can be viewed as a percent of whole code base covered by tests. This metric has its pros and cons. One of its advantages is that it shows how well are classes structured and responsibilities split between classes. High test coverage means that project architecture has low coupling and classes have their specific roles in the system. This would lead to better understanding of the system by the developer, and would make it easier to change the system. Which leads to higher maintainability.

However, this way of evaluating architecture may be misleading on its own. Tests may not be good and may be misleading. High test coverage as the main goal of a project architecture leads to focusing more on testing, but not on achieving the desired behavior. And implementing the right behavior of architectural components is the main goal of any architecture.

Test coverage is collected as a combination of unit-test coverage and ui-test coverage. Unit tests is implemented using Junit and Mockito frameworks. UI tests are implemented using Espresso testing framework. Reports combined by means of Jacoco tool.

To measure modifiability, all use cases implementation was split into features. Features are the units of software project development. Number of features is one of the metrics that describes the comlexity of the project. These features are following:

- infrastructure layer (database, dao);
- domain layer (domain models and abstractions);
- base architecture (view layer models and abstractions)
- main screen (main view container)
- create note (create note screen, insert note interactor, router);
- view note (note screen, read note interactor, router);
- edit note (edit note screen, update note interactor, router);
- delete note (delete note interactor, router);
- view list of all notes (notes list screen, read all notes interactor, router);
- create category (create category screen, insert category interactor, router);
- edit category (edit category screen, update category interactor, router);
- delete category (delete category interactor, router);
- set category for a note (category selection element, update note interactor);

- view notes by category (categories screen, read all categories interactor, router).

5.2 Results evaluation

Modifiability metrics are collected for every feature. These metrics are number of classes that need to be created to implement feature, number of classes that need to be modified to implement feature and number of lines of code that need to be added or modified to implement feature.

Feature	VIPER	MVP
Infrastructure layer	5 classes	5 classes
Domain layer	5 classes	5 classes
Base architecture	21 classes	24 classes
Main screen	8 classes	4 classes
Create note	9 classes	5 classes
View note	7 classes	7 classes
Edit note	2 classes	0 classes
Delete note	2 classes	0 classes
View list of all notes	10 classes	6 classes
Create category	9 classes	5 classes
Edit category	9 classes	5 classes
Delete category	2 classes	0 classes
Set category for a note	1 class	1 class
View notes by category	10 classes	6 classes

Table 5.1: Number of classes that need to be created to implement feature.

Table 5.1 shows number of classes that need to be added to implement each particular feature for both architectures. VIPER architecture requires to create more classes to implement each specific feature, then MVP.

Feature	VIPER	MVP
Infrastructure layer	0 classes	0 classes
Domain layer	0 classes	0 classes
Base architecture	0 classes	0 classes
Main screen	4 classes	4 classes
Create note	7 classes	8 classes
View note	4 classes	6 classes
Edit note	4 classes	5 classes
Delete note	5 classes	6 classes
View list of all notes	5 classes	5 classes
Create category	5 classes	8 classes
Edit category	5 classes	6 classes
Delete category	4 class	5 classes
Set category for a note	6 class	6 class
View notes by category	8 classes	9 classes

Table 5.2: Number of classes that need to be edited to implement feature.

Table 5.2 shows number of classes that need to be edited to implement each particular feature for both architectures. VIPER architecture requires to edit less classes to implement each specific feature, than MVP. This is explained by higher number of classes with well defined responsibilities and lesser coupling between the components.

Feature	VIPER	MVP
Infrastructure layer	216 LOC	216 LOC
Domain layer	270 LOC	270 LOC
Base architecture	631 LOC	659 LOC
Main screen	242 LOC	185 LOC
Create note	356 LOC	336 LOC
View note	278 LOC	266 LOC
Edit note	102 LOC	74 LOC
Delete note	110 LOC	80 LOC
View list of all notes	293 LOC	267 LOC
Create category	336 LOC	300 LOC
Edit category	390 LOC	344 LOC
Delete category	51 LOC	35 LOC
Set category for a note	123 LOC	107 LOC
View notes by category	452 LOC	405 LOC

Table 5.3: Number of lines of code that need to be added or edited to implement feature.

Table 5.3 shows number of lines of code that need to be added or edited to implement each particular feature for both architectures. VIPER architecture requires to write 10% more lines of code on average to implement each specific feature, than MVP. Full tables with lists of classes can be seen in appendix.

Performance results were collected using automated testing framework Espresso for Android. Typing and clicking controls speed was constant, as these were performed by automated framework. Drawing and loading speed was constant, as device setup was constant. Only factor affecting time of execution and memory allocation was amount of operations that needed to be loaded, which is different for application architectures. 1000 measurements were made to calculate average to make sure error is insignificant. All measurements were split into two tests, that consisted of 500 execution times.

Nexus 5X 8.1.0, time(ms)	Create note	Edit Note	Create category	Edit category	Create category note	Edit category note	Delete note	Delete category	Delete category note	Show info
Viper test 1	2920	2766	2320	2453	3554	2740	1401	1600	1824	1057
Mvp test 1	3142	3000	2479	2651	3774	2935	1534	1690	2011	1100
Viper test 2	2912	2726	2288	2519	3574	2762	1443	1622	1884	1095
Mvp test 2	3182	2992	2471	2711	3752	2905	1500	1688	2033	1186
Viper avg.	2916	2746	2309	2486	3564	2751	1422	1611	1854	1076
Mvp avg.	3162	2996	2475	2681	3763	2920	1517	1689	2027	1143

Table 5.3: Execution time of use cases.

Table 5.3 shows execution time for each use case implemented with both architectures. Use cases were executed in stable environment 1000 times and table shows average. This number of test executions is selected to reduce the effect of possible errors or performance spikes of the environment. Two tests were conducted, each consisted of 500 execution times.

Nexus 5X 8.1.0, memory (mb)	Create note	Edit Note	Create category	Edit category	Create category note	Edit category note	Delete note	Delete category	Delete category note	Show info
Viper test 1	120.9	105.3	86.6	103.5	129.0	104.5	80.5	83.9	85.3	59.8
Mvp test 1	112.1	121.2	110.1	112.7	142.1	120.0	88.2	100.1	99.8	66.0
Viper test 2	120.3	107.3	87.8	98.7	127.6	122.5	80.3	82.1	89.1	61.8
Mvp test 2	108.1	119.6	115.5	132.7	148.1	129.4	92.2	88.1	99.6	67.0
Viper avg.	120.6	106.3	87.2	101.2	128.3	113.5	80.4	83.0	87.2	60.8
Mvp avg.	110.1	120.4	112.8	122.7	145.1	124.7	90.2	94.1	99.7	66.5

Table 5.4: Memory allocation for use cases.

Table 5.4 shows memory allocation for each use case implemented with both architectures. Memory allocation was measured with Android profiler. This tool was implemented to measure performance properties of applications.

Testability results can be seen in figures 1 and 2 in appendix. They show test coverage reports for VIPER and MVP projects containing all use cases. MVP and VIPER have test coverage of 85%. This means that VIPER and MVP are both well testable and there is no difference between these two architectures. The reason is that both architectures describe components of the system that should not be big. Both these architectures aim to provide high cohesion for components and good setup for unit testing.

Collected metrics on modifiability show that VIPER architecture requires more classes to be created, but less classes to be edited to implement every feature. Every class represents one component described by the architecture. Components can be split into more classes, but the rule “one component – one class” was designed to make modifiability metrics meaningful. This shows that VIPER architecture classes are smaller and less complex. This leads to better understanding of classes.

VIPER architecture requires more classes to be created because there are more different components with different responsibilities. Every feature with newly displayed screen requires to create the new view, presenter and router contracts as well as implementations for those. For every feature, that brings new business logic rules to the application, new interactor contract and implementation have to be created. New features also bring changes to existing routers.

As for MVP, every new feature with newly displayed screen requires new view and presenter contracts as well as implementations for those to be created. New business logic rules mostly bring changes to the existing facades and don’t require new classes to be created.

However, MVP architecture requires less lines of code to be added for every feature. This means that it will take less time to implement new feature in MVP

architecture. This is particularly important for projects with strict deadlines and huge amount of work required to be done, like startups, for example.

VIPER requires more lines of code due to a higher amount of classes and components, that combined create an architecture. Every class, apart from the actual algorithms, describe dependencies and relations of this class. These descriptions may repeat from class to class, and this is because a total number of lines of code, in this case, is higher. MVP in the business logic layer mostly rely on big facades, that contain a lot of methods. But the dependencies in these facades usually don't repeat.

VIPER architecture shows better performance results. Application built using VIPER architecture takes less time to go through application use case scenarios, than MVP. VIPER architecture needs less memory to be allocated for application by operating system, than MVP.

Better performance results of the VIPER architecture are achieved because of the smaller size of classes. Smaller classes need less memory to be loaded into the memory. Business logic, executed in test use case scenarios is the same for both architectures, so the number of instructions is almost the same as well. But, due to MVP software architecture having heavier classes, they require more time to be loaded in memory. This explains the fact, that test use case scenarios developed using MVP software architecture take more time to be executed.

RQ1. Which Android Architecture is more maintainable, MVP or VIPER?

Maintainability is evaluated as composition of testability and modifiability. VIPER architecture showed better results in testability, but difference was insignificant. MVP architecture required smaller amount of LOC to implement each feature by 10% on average.

A number of lines of code that need to be added or modified are the metric that is less important for modifiability. A higher number of classes in the VIPER software architecture, than in MVP architecture is actually a good thing. A higher number of classes means that they have a smaller set of responsibilities. This means that finding a possible bug in the software system is easier. Big classes are harder to understand and explore.

However, a lesser amount of lines of code, that need to be added or modified to implement a feature, means that it is faster to implement a feature this way. MVP software architecture allows the development team to implement a project faster, than if they are using VIPER software architecture.

RQ2. Which Android Architecture leads to better performance, MVP or VIPER?

Performance is evaluated by measuring time, required to finish use case scenario, and memory consumption. VIPER performed better on both of these metrics. VIPER showed better results in both performance measurements in the test environment. Test environment was an Android device with middle range hardware powers. These results may vary on other devices. The difference is less significant on more powerful devices. If the main market of the android application project is more powerful flagship devices, then performance difference should not be the main criteria to select the architecture. However, if the main market of the android application project also relies on less powerful devices, the difference in memory consumption may become crucial for big and complex applications. Mostly, the difference is not significant, as mostly mobile applications are not big and recent Android devices of all price ranges have powerful hardware.

RQ3. Which Android Architecture fits better for different types of projects, MVP or VIPER?

MVP architecture requires less lines of code to be written, than VIPER, so it will take less time for developer to complete application. This leads to lower cost as well. VIPER architecture shows better performance and testability, so this architecture will be preferable if application performance and quality are more important than cost and time.

MVP architecture will be better than VIPER for small projects, as performance difference will not be visible, but cost and time consumption will be lower. VIPER architecture will be better than MVP for medium to big scaled projects, as performance will be a significant issue when lots of business logic is performed. VIPER architecture leads to cleaner code, than MVP, as classes are smaller and responsibilities are separated better between classes. VIPER architecture will fit better for modular applications, as these applications are big, have difficult structure and depends a lot on performance. MVP architecture will suit better for libraries, as resulting size of the library is important, and MVP project will have less lines of code. Size is important for libraries, as projects usually use a lot of them, and these projects tent to become very heavy, due to big size of frameworks used.

5.3 Summary

This chapter reveals details about the experiment. It starts with the description of the metrics that need to be collected and motivates this selection. These metrics are modifiability, testability, and performance. These metrics are collected for both architectures on test use case scenarios. Test use case scenarios are executed in a stable environment. The first part of this chapter is dedicated to the explanation of the details of the experiment. It has the description of all the test use case scenarios. Then it describes the metrics, that are selected and explains the way, these metrics are collected.

The second part of the chapter contains the results of the experiment. It shows the tables with the results of measurements. VIPER architecture showed better results on both performance metrics. Testability of MVP and VIPER architecture is on the same level. Modifiability results showed that both MVP and VIPER have advantages in different measurements. Then, research questions are answered based on the measurements. The results show that both architectures are viable and are better suitable for different types of software development projects.

6 Conclusions and Future Work

The thesis aims to define whether new VIPER architecture is better than more conservative MVP architecture. To achieve this thesis starts with studying the basic concepts of architecture design and learning the structure of both architectures to define similarities and differences.

The literature review showed that research on the matter of android architecture comparison is present. This research describes different approaches to compare architectures. These approaches aim to evaluate architectures in a few key metrics and then analyze the results.

Architecture comparison starts with the definition of three main properties. These properties are maintainability, performance, and testability. Each property is then explained by means of establishing criteria to evaluate the quality. In the end, all measurements are compared and analyzed.

Modifiability assessment consists of three key measurements. A number of classes that need to be created to implement a feature, number of classes that need to be modified to implement a feature and number of lines of code that need to be added or modified to implement a feature are measurements to evaluate modifiability. VIPER required more classes to be created to develop a feature, while fewer classes to be modified than MVP architecture. VIPER required 10% more lines of code to be added or modified to implement a feature.

Memory consumption and time required to run use case scenario are two measurements to evaluate the performance of the architecture. Running test scenarios was fully automated to ensure that factors other then architecture features did not affect results. VIPER showed better results on both measurements. Use cases implemented using MVP architecture required more time to be executed and consumed more memory.

Testability is evaluated by means of measuring test coverage. Test coverage is measured by combining both unit, integration and user interface tests. VIPER architecture and MVP showed the same results.

6.1 Conclusions

In conclusion, both architectures are viable. Set of metrics was collected for both architectures, describing their properties, such as modifiability, testability, maintainability and performance. These metrics shown that one architecture has better in one metrics, but worse in other. This means that there is no absolute best solution for every situation and architecture should always be selected according to project needs and restrictions. These restrictions may be concerning cost or time of development, performance of application or quality of code. Projects may also have very different nature, like mobile application with user interface, or some framework for mobile development. All these differences affect which architecture to select for project.

This thesis provides information for two popular android project architectures MVP and VIPER. Selection of right architecture for the project will be easier, when based on this research. This is especially useful in industry, as amount of new android applications, frameworks and libraries is increasing constantly.

Results provided by this research are not applicable to other areas of software development. But methods developed in this thesis to compare two mobile architectures can be used to compare any other architectures, not just for mobile

applications, but for other types of projects as well. This method can be used to create a benchmark of mobile architectures.

Which Android architecture is more maintainable, MVP or VIPER? Collected metrics show that MVP requires less classes to be added to implement a new feature, but more classes to be edited. Difference in testability metric was insignificant. The results show that there is no direct answer to this question and all these metrics should be taken into account when selecting the architecture.

Which Android architecture leads to better performance, MVP or VIPER? VIPER architecture had better results in both performance metrics. This means that Viper architecture leads to better performance than MVP architecture.

Which Android architecture fits better for different types of projects, MVP or VIPER? MVP architecture would fit better for small scaled projects and libraries. VIPER architecture provides advantages for medium and big projects, as well as for modular projects.

6.2 Future work

Future work may include more architecture metrics to compare. There are metrics that can represent intra-class cohesion and coupling between classes. These metrics can also be calculated for modules. Comparison on these metrics may give more information on the subject of maintainability and complexity of a project that is implemented using specific architecture.

There could have been more use cases implemented to investigate architectures more. Future work may include architecture comparison on very big and complex use cases.

Collecting metrics, mentioned in this thesis, on big commercial projects will improve the accuracy of the results. It will also allow creating a stable benchmark for Android architectures.

Measuring performance on a large variety of Android devices will greatly improve the understanding of both architectures and help to define ways to improve performance for these architectures. It may also be a step to create a new architecture that will solve the issues of MVP and VIPER and greatly influence android application design.

Android architecture design is relatively new and active field of knowledge, so new architectures will appear every few years. Architecture comparison will be an issue and methods described in this thesis may be useful. Comparing new architectures with more conservative ones is the point of future research.

7 References

[1] Tian Lou. *A Comparison of Android Native App Architecture – MVC, MVP and MVVM*. Helsinki: Aalto University, Master's Thesis, 2016.

[2] Anthony Madhvani. *Structuring Android Applications*. Kortrijk: The Hogeschool West Vlaanderen College, Bachelor's Thesis, 2016.

[3] Hugo Kallstrom. *Increasing Maintainability of Android Applications.* Umea : Umea University, Department of Computing Science, Master's Thesis, 2016.

[4] Jussi Koskinen. *Software maintenance costs*. Jyvaskyla: University of Jyvaskyla, 2010.

[5] Douglas Gilstrap. "Ericsson mobility report on the pulse of the networked society," *Ericsson*, 2016. [Online]. Available: https://www.slideshare.net/EricssonLatinAmerica/ericsson-mobility-report-november-2016. [Accessed: 25.01.2018].

[6] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating software architectures*. Boston: Addison-Wesley, 2003.

[7] International Organization of Standardization, "ISO/IEC 25010:2011", *International Organization of Standardization,* 2011. [Online]. Available: https://www.iso.org. [Accessed: 20.01.2018].

[8] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. London: Pearson Education, 2009.

[9] A. Patidar and U. Suman, "A survey on software architecture evaluation methods" in *Computing for Sustainable Global Development (INDIACom)*. 2nd International Conference on, Indore, 2015.

[10] L. Chen, M. A. Babar and B. Nuseibeh, "Characterizing Architecturally Significant Requirements," in *IEEE Software*, vol. 30, no. 2, pp. 38 - 45, 2013.

[11] M. E. Joorabchi, V. B. C. Univ. of British Columbia, A. Mesbah and P. Kruchten, "Real Challenges in Mobile App Development," in *Empirical Software Engineering and Measurement*, 2013 ACM/IEEE International Symposium on, 2013.

[12] M. Mattsson, H. Grahn and F. Mårtensson, "Software architecture evaluation methods for performance, maintainability, testability, and portability," in *Second International Conference on the Quality of Software Architectures*, 2006.

[13] Douglas Gilstrap. "Ericsson mobility report on the pulse of the networked society," *Ericsson*, 2015. [Online]. Available: https://www.slideshare.net/Ericsson/ericsson-mobilityreportjune2015. [Accessed: 25.01.2018].

[14] Mahfuzul Huda, Dr. Y.D.S. Arya, and Dr. M.H. Khan. “Measuring testability of object oriented design: A systematic review,” *International Journal of Scientific Engineering and Technology* vol. 3 no.10, pp1313-1319, 2014.

[15] Himanshu Shewale, Sameer Patil, Vaibhav Deshmukh, Pragya Singh. Analysis of Android vulnerabilities and modern exploitation techniques. *ICTACT journal on communication technology*, march 2014.

[16] M. Rizwan Jameel Qureshi, Fatima Sabir. “A comparison of model view controller and model view presenter,” *Science International (Lahore)*, vol. 25(1), pp. 7-9, 2013.

[17] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, Muhammad Ali Babar. “10 years of software architecture knowledge management: Practice and future,” *Journal of Systems and Software*, vol. 116, pp. 191-205, June 2016.

[18] GitHub, Inc. “Robotium Android testing framework.” [Online]. Available: https://github.com/RobotiumTech/robotium. [Accessed: 25.01.2018]

[19] Szczepan Faber, Brice Dutheil, Rafael Winterhall. “Mockito Android testing framework.” [Online]. Available: http://site.mockito.org/. [Accessed: 25.01.2018]

[20] Google LLC, “Official Android dashboards.” [Online]. Available: https://developer.android.com/about/dashboards/. [Accessed: 25.02.2018]

A Appendix 1

Table 1: Number and list of classes, that need to be created to implement particular feature

Feature	VIPER	MVP
Infrastructure layer	5 classes: CategoryDao.java, NoteDao.java, DbCategory.java, DbNote.java, NotesDatabase.java	5 classes: CategoryDao.java, NoteDao.java, DbCategory.java, DbNote.java, NotesDatabase.java
Domain layer	5 classes: Category.java, Note.java, CategoryDbMapper.java, NoteDbMapper.java, CategoryUtil.java	5 classes: Category.java, Note.java, CategoryDbMapper.java, NoteDbMapper.java, CategoryUtil.java
Base architecture	21 classes: BaseMapper.java, UserScope.java, ApplicationComponent.java, ActivityComponent.java, ActivityModule.java, ApplicationModule.java, DbModule.java, InteractorModule.java, MapperModule.java, PresenterModule.java, RouterModule.java, SchedulerModule.java, BaseAdapter.java, BaseViewHolder.java, ItemLongClickSelectedListener.java, ItemSelectedListener.java, BaseActivity.java, BaseFragment.java, BaseRouter.java, RxPresenter.java, NotesApplication.java	24 classes: BaseMapper.java, UserScope.java, ApplicationComponent.java, ActivityComponent.java, ActivityModule.java, ApplicationModule.java, DbModule.java, RepositoryModule.java, MapperModule.java, PresenterModule.java, SchedulerModule.java, BaseAdapter.java, BaseViewHolder.java, ItemLongClickSelectedListener.java, ItemSelectedListener.java, BaseActivity.java, BaseFragment.java, BaseRouter.java, RxPresenter.java, NotesApplication.java, CategoryRepository.java, CategoryDbRepository.java NoteRepository.java, NoteDbRepository.java
Main screen	8 classes: MainPresenter.java, MainPresenterImpl.java, MainRouter.java, MainRouterImpl.java, MainView.java, MainActivity.java, InitDbInteractor.java, InitDbInteractorImpl.java	4 classes: MainPresenter.java, MainPresenterImpl.java, MainView.java, MainActivity.java
Create note	9 classes:	5 classes:

	CreateNotePresenter.java, CreateNotePresenterImpl.java, CreateNoteRouter.java, CreateNoteRouterImpl.java, CreateNoteFragment.java, CreateNoteView.java, CreateNoteActivity.java, AddNoteInteractor.java, AddNoteDbInteractor.java	CreateNotePresenter.java, CreateNotePresenterImpl.java, CreateNoteFragment.java, CreateNoteView.java, CreateNoteActivity.java,
View note	7 classes: EditNotePresenter.java, EditNotePresenterImpl.java, EditNoteRouter.java, EditNoteRouterImpl.java, EditNoteFragment.java, EditNoteView.java, EditNoteActivity.java	7 classes: EditNotePresenter.java, EditNotePresenterImpl.java, EditNoteFragment.java, EditNoteView.java, EditNoteActivity.java
Edit note	2 classes: EditNoteInteractor.java, EditNoteDbInteractor.java	0 classes
Delete note	2 classes: DeleteNoteInteractor.java, DeleteNoteDbInteractor.java	0 classes
View list of all notes	10 classes: NotesListPresenter.java, NotesListPresenterImpl.java, NotesListRouter.java, NoteslistRouterImpl.java, NotesListFragment.java, NotesListView.java, NotesAdapter.java, NotesViewHolder.java, NotesListInteractor.java, NotesListDbInteractor.java	6 classes: NotesListPresenter.java, NotesListPresenterImpl.java NotesListFragment.java, NotesListView.java, NotesAdapter.java, NotesViewHolder.java
Create category	9 classes: CreateCategoryPresenter.java, CreateCategoryPresenterImpl.java, CreateCategoryRouter.java, CreateCategoryRouterImpl.java, CreateCategoryView.java, CreateCategoryFragment.java, CreateCategoryActivity.java, AddCategoryInteractor.java, AddCategoryDbInteractor.java	5 classes: CreateCategoryPresenter.java, CreateCategoryPresenterImpl.java, CreateCategoryView.java, CreateCategoryFragment.java, CreateCategoryActivity.java
Edit category	9 classes: EditCategoryPresenter.java, EditCategoryPresenterImpl.java, EditCategoryRouter.java, EditCategoryRouterImpl.java, EditCategoryView.java, EditCategoryFragment.java, EditCategoryActivity.java, EditCategoryInteractor.java,	5 classes: EditCategoryPresenter.java, EditCategoryPresenterImpl.java, EditCategoryView.java, EditCategoryFragment.java, EditCategoryActivity.java

	EditCategoryDbInteractor.java	
Delete category	2 classes: DeleteCategoryInteractor.java, DeleteCategoryDbInteractor.java	0 classes
Set category for a note	1 class: CategoryAdapter.java	1 class: CategoryAdapter.java
View notes by category	10 classes: CategoriesListPresenter.java, CategoriesListPresenterImpl.java, CategoriesListRouter.java, CategorieslistRouterImpl.java, CategoriesListFragment.java, CategoriesListView.java, CategoryAdapter.java, CategoryViewHolder.java, CategoriesListInteractor.java, CategoriesListDbInteractor.java	6 classes: CategoriesListPresenter.java, CategoriesListPresenterImpl.java, CategoriesListFragment.java, CategoriesListView.java, CategoryAdapter.java, CategoryViewHolder.java

Table 2: Number and list of classes, that need to be edited to implement particular feature

Feature	VIPER	MVP
Infrastructure layer	0 classes	0 classes
Domain layer	0 classes	0 classes
Base architecture	0 classes	0 classes
Main screen	4 classes: RouterModule.java, PresenterModule.java, ActivityComponent.java, InteractorModule.java	4 classes: PresenterModule.java, ActivityComponent.java, CategoriesRepository.java, CategoriesDbRepository.java
Create note	7 classes: RouterModule.java, PresenterModule.java, ActivityComponent.java, InteractorModule.java, NotesListRouterImpl.java, NotesListPresenter.java, NotesListPresenterImpl.java	8 classes: PresenterModule.java, ActivityComponent.java, NotesRepository.java, NotesDbRepository.java, NotesListView.java, NotesListFragment.java, NotesListPresenter.java, NotesListPresenterImpl.java
View note	4 classes: RouterModule.java, PresenterModule.java, ActivityComponent.java,	6 classes: PresenterModule.java, ActivityComponent.java, NotesListFragment.java,

	NotesListRouterImpl.java	NotesListView.java, NotesListPresenter.java, NotesListPresenterImpl.java
Edit note	4 classes: EditNotePresenterImpl.java, EditNoteFragment.java, EditNoteView.java, InteractorModule.java	5 classes: EditNotePresenterImpl.java, EditNoteFragment.java, EditNoteView.java, NotesRepository.java, NotesDbRepository.java
Delete note	5 classes: EditNotePresenterImpl.java, EditNotePresenter.java, EditNoteFragment.java, EditNoteView.java, InteractorModule.java	6 classes: EditNotePresenterImpl.java, EditNotePresenter.java, EditNoteFragment.java, EditNoteView.java, NotesRepository.java, NotesDbRepository.java
View list of all notes	5 classes: MainRouterImpl.java, PresenterModule.java, RouterModule.java, ActivityComponent.java InteractorModule.java	5 classes: PresenterModule.java, ActivityComponent.java, NotesRepository.java, NotesDbRepository.java, ActivityComponent.java
Create category	5 classes: RouterModule.java, PresenterModule.java, ActivityComponent.java, InteractorModule.java, CategoriesListRouter.java	8 classes: PresenterModule.java, ActivityComponent.java, CategoriesListFragment.java, CategoriesListView.java CategoriesRepository.java, CategoriesDbRepository.java, CategoriesListFragment.java, CategoriesListView.java
Edit category	5 classes: RouterModule.java, PresenterModule.java, ActivityComponent.java, InteractorModule.java, CategoriesListRouterImpl.java	6 classes: PresenterModule.java, ActivityComponent.java, CategoriesRepository.java, CategoriesDbRepository.java, CategoriesListFragment.java, CategoriesListView.java
Delete category	4 class: InteractorModule.java, EditCategoryFragment.java, EditCategoryPresenter.java, EditCategoryPresenterImpl.java	5 classes: EditCategoryFragment.java, EditCategoryPresenter.java, EditCategoryPresenterImpl.java, CategoriesRepository.java, CategoriesDbRepository.java
Set category for a note	6 class: CreateNoteFragment.java, CreateNotePresenter.java, CreateNotePresenterImpl.java, EditNotePresenter.java, EditNotePresenterImpl.java, EditNoteFragment.java	6 class: CreateNoteFragment.java, CreateNotePresenter.java, CreateNotePresenterImpl.java, EditNotePresenter.java, EditNotePresenterImpl.java, EditNoteFragment.java

View notes by category	8 classes: MainRouterImpl.java, PresenterModule.java, RouterModule.java, ActivityComponent.java, InteractorModule.java, NotesListFragment.java, NotesListPresenter.java, NotesListPresenterImpl.java	9 classes: MainActivity.java, PresenterModule.java, RouterModule.java, ActivityComponent.java, NotesListFragment.java, NotesListPresenter.java, NotesListPresenterImpl.java, NotesRepository.java, NotesDbRepository.java

Table 3: Lines of code that need to be added/edited to implement particular feature

Feature	VIPER	MVP
Infrastructure layer	216 LOC	216 LOC
Domain layer	270 LOC	270 LOC
Base architecture	631 LOC	659 LOC
Main screen	242 LOC	185 LOC
Create note	356 LOC	336 LOC
View note	278 LOC	266 LOC
Edit note	102 LOC	74 LOC
Delete note	110 LOC	80 LOC
View list of all notes	293 LOC	267 LOC
Create category	336 LOC	300 LOC
Edit category	390 LOC	344 LOC
Delete category	51 LOC	35 LOC
Set category for a note	123 LOC	107 LOC
View notes by category	452 LOC	405 LOC

Figure 1: test coverage report for MVP

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
masters.vlad.humeniuk.notesmvp.database		34%		40%	10	22	34	68	2	12	0	3
masters.vlad.humeniuk.notesmvp.di.modules		77%		n/a	34	97	49	206	34	97	3	25
masters.vlad.humeniuk.notesmvp.database.dao		88%		64%	8	40	8	176	0	29	0	8
masters.vlad.humeniuk.notesmvp.presentation.base		77%		58%	14	34	21	78	10	28	0	4
masters.vlad.humeniuk.notesmvp.di.components		90%		50%	6	48	7	144	0	42	0	6
masters.vlad.humeniuk.notesmvp.presentation.main		74%		60%	6	18	9	45	4	15	0	3
masters.vlad.humeniuk.notesmvp.presentation.editnote.view		91%		60%	6	25	7	79	4	22	0	3
masters.vlad.humeniuk.notesmvp.presentation.editcategory.view		90%		60%	5	24	7	76	3	21	0	4
masters.vlad.humeniuk.notesmvp.presentation.noteslist.view		90%		75%	6	26	8	68	4	22	0	5
masters.vlad.humeniuk.notesmvp.presentation.createcategory.view		89%		50%	5	24	7	69	3	22	0	4
masters.vlad.humeniuk.notesmvp.presentation.createnote.view		87%		50%	5	21	7	59	3	19	0	3
masters.vlad.humeniuk.notesmvp.presentation.categories.view		90%		50%	5	24	6	61	3	22	0	5
masters.vlad.humeniuk.notesmvp.presentation.categories.view.list		79%		0%	2	10	7	29	1	9	0	3
masters.vlad.humeniuk.notesmvp.presentation.noteslist.view.list		77%		0%	2	9	7	26	1	8	0	3
masters.vlad.humeniuk.notesmvp.domain.entity		93%		50%	10	42	4	67	4	36	0	4
masters.vlad.humeniuk.notesmvp.presentation.createnote		91%		n/a	1	7	1	14	1	7	0	1
masters.vlad.humeniuk.notesmvp.presentation.editcategory.presenter		97%		75%	1	11	2	30	0	9	0	1
masters.vlad.humeniuk.notesmvp.presentation.createcategory.presenter		96%		75%	1	8	2	23	0	6	0	1
masters.vlad.humeniuk.notesmvp.presentation.base.list		95%		n/a	1	9	1	20	1	9	0	2
masters.vlad.humeniuk.notesmvp.domain.utils		83%		n/a	1	2	1	6	1	2	0	1
masters.vlad.humeniuk.notesmvp.database.entity		99%		50%	11	33	0	40	0	22	0	2
masters.vlad.humeniuk.notesmvp.domain.mappers		98%		50%	1	7	0	30	0	6	0	2
masters.vlad.humeniuk.notesmvp.domain.repositories.implementation		100%		100%	0	33	0	69	0	29	0	2
masters.vlad.humeniuk.notesmvp.presentation.editnote.presenter		100%		75%	1	12	0	38	0	10	0	1
masters.vlad.humeniuk.notesmvp.presentation.createnote.presenter		100%		83%	1	11	0	35	0	8	0	1
masters.vlad.humeniuk.notesmvp.presentation.categories.presenter		100%		50%	1	7	0	19	0	6	0	1
masters.vlad.humeniuk.notesmvp.presentation.noteslist.presenter		100%		n/a	0	6	0	18	0	6	0	1
masters.vlad.humeniuk.notesmvp.presentation.editcategory		100%		n/a	0	6	0	13	0	6	0	1
masters.vlad.humeniuk.notesmvp.presentation.editnote		100%		n/a	0	6	0	13	0	6	0	1
masters.vlad.humeniuk.notesmvp.presentation.createcategory		100%		n/a	0	6	0	11	0	6	0	1
masters.vlad.humeniuk.notesmvp.presentation		100%		n/a	0	3	0	8	0	3	0	1
masters.vlad.humeniuk.notesmvp.presentation.main.presenter		100%		n/a	0	2	0	8	0	2	0	1
masters.vlad.humeniuk.notesmvp.presentation.createnote.view.spinner		100%		n/a	0	1	0	2	0	1	0	1
Total	910 of 6,014	85%	72 of 169	57%	144	634	195	1,648	79	548	3	105

Figure 2: test coverage results for VIPER