UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Mustafa Ogün Öztürk

Evaluating Maintainability of Android Applications: Mooncascade Case Study

Master's Thesis (30 ECTS)

Supervisor: Jakob Mass, MSc

Tartu 2021

Evaluating Maintainability of Android Applications: Mooncascade Case Study

Abstract:

Abstract

Keywords:

List of keywords

CERCS:

CERCS code and name: `https://www.etis.ee/Portal/Classifiers/Details/d3717f7b-bec8-4cd9-8ea4-c89cd56ca46e`

Tüübituletus neljandat järku loogikavalemitele

Lühikokkuvõte:

One or two sentences providing a basic introduction to the field, comprehensible to a scientist in any discipline.

Two to three sentences of more detailed background, comprehensible to scientists in related disciplines.

One sentence clearly stating the general problem being addressed by this particular study.

One sentence summarising the main result (with the words “here we show´´ or their equivalent).

Two or three sentences explaining what the main result reveals in direct comparison to what was thought to be the case previously, or how the main result adds to previous knowledge.

One or two sentences to put the results into a more general context.

Two or three sentences to provide a broader perspective, readily comprehensible to a scientist in any discipline, may be included in the first paragraph if the editor considers that the accessibility of the paper is significantly enhanced by their inclusion.

Võtmesõnad:

List of keywords

CERCS:

CERCS kood ja nimetus: `https://www.etis.ee/Portal/Classifiers/Details/d3717f7b-bec8-4cd9-8ea4-c89cd56ca46e`

Contents

Unsolved issues

1 Introduction

In the last decade, the impact of smartphones on our lives has significantly increased, and smartphones and mobile applications became people's primary way of interacting with technology. This situation has made the applications that work on these smartphones a vital part of daily and business life, from ordinary people to large companies. Today, mobile applications have become one of the most critical parts of digitalisation. Notably, as a successful open-source mobile operating system, Android has been a core element of this change, and the demand for Android applications has increased. Android application development has become one of the most necessary parts of the business area with a significant market share. Today, there are more than 2.5 billion active Android devices in the world [1] . Therefore, it is difficult to ignore the importance of Android application development, considering the Android operating system's market share and many people's interactions with mobile applications running on Android devices. Consequently, Android application development has become an essential topic in the IT industry and academy. However, the increasing importance of the mobile era also brought more challenges to mobile application development and, of course, to Android application development.

Today, the difficulties that arise during Android application development can be examined under four major topics. These are Android's nature and its platform-specific components, demanding business requirements and sophisticated business needs, the frequent update rate of Android applications, and lastly, growing codebases and fast-changing development teams. When developing Android applications, it is essential to facilitate these difficulties. To overcome these difficulties, "maintainability" emerges as one of the most important non-functional requirements when developing Android applications. While developing Android applications, it is wise to use the technologies and techniques to increase the Android application's maintainability. Developing high-quality applications that can survive in the competitive Android market and achieving these goals in a time and cost-effective manner is only possible in this way.

This study's primary purpose is to identify and explain the methodologies and technologies used by Mooncascade, a leading software product development company with global reach, to develop real-world enterprise Android applications and evaluate these methodologies and technologies in terms of maintainability. To that aim, both qualitative analyses (in the forms of interviews, questionnaires for measuring the impact from the software complexity point of view) and quantitative analysis (measuring maintainability via object oriented software quality metrics) will be conducted.

Considering the scarcity of academic studies in this field and the inadequacy of these studies' content, and actuality problems, it can be said that this study will make a significant contribution to the academic studies in the Android field. The inadequacy

and not being up-to-date with academic studies in the Android field are among the main motivations for creating this study. Another important source of motivation for this study is the negative effects of the problems arising from the lack of maintainability in Android application development on the developer experiences and the elimination of these negative effects and thus, to improve Android developer experience.

The target audience of this study includes Android developers and researchers who are already experienced with Android application development basics and willing to learn advanced techniques and tools for Android application development. The study facilitates developers and researchers to follow the most up-to-date software engineering and Android development practices to develop state-of-the-art Android applications with high maintainability. The study only focuses on native Android application development.

1.1 Problem Statement

Mooncascade is a leading product development company with global reach, based in Estonia. Mooncascade aims to provide quality at every stage of the software product development process, helping clients build innovative solutions that inspire, disrupt, and challenge markets worldwide. The company provides various software development services, including web development, data science, quality assurance, and mobile application development.

The Android development team is one of the most active units of Mooncascade. The team has been working with clients from different industry domains and developing countless Android applications for clients. From the client's point of view, Mooncascade's Android team's biggest goal is developing Android apps with high quality. High quality means, following the latest industry trends and technologies, improving and providing a pleasant mobile application experience for customers and end-users. From the software engineering perspective, essential goals are enhancing developer and team efficiency and increasing code maintainability. These are the goals that will constitute this study's main subject since they are directly related to the four significant challenges encountered when developing Android applications, which are mentioned in the introduction section. To have a better understanding of those challenges, it is wise to explain them in a bit more detail.

Android platform-specific complexity: Android applications are distinguished from traditional web and desktop software with their sophisticated features and specific structure. They need to fulfill some platform-specific requirements. Apps should work per Android OS, and they have to use components offered by the Android Software Development Kit, such as Activity, Fragment, Service. A typical Android app consists of components such as views, activities, fragments, services, broadcast receivers, and content providers. These components are unique to the Android platform and provided

by the Android SDK. The Android system mostly controls the behaviors of these components. Android developers must obey the contract supplied by the system when using these components, and this situation sometimes limits developers to use some specific programming techniques when developing Android applications. These components are directly related to the Android OS, and they should be separated from the other possible layers to decrease the level of complexity and increase maintainability. Nevertheless, that is not always the case, and sometimes this separation is not as easy as it sounds. Besides, there are other components used in an Android app that are not a part of the Android SDK for managing network operations, database transactions, and business rules. Whether it is a part of the Android SDK or not, each of these components represents a different "concern" of an Android application. The necessity of working together in harmony for all these components, the challenges that arise from the Android operating system and Android SDK's nature, and the limited resources of mobile devices make Android applications complicated software systems that are hard to develop.

Business-specific complexity: Android applications get more and more sophisticated to fulfill increasing user needs and business requirements. Mobile applications become more functional as user and business needs increase. Consequently, the complexity of Android apps from the software development point of view increases, and apps become more business-critical [2]. When this business-specific complexity comes together with the platform-specific complexity mentioned above, it makes developers' jobs even harder as its influence on development is strong. In this regard, it makes sense to separate the business-specific requirements and logic from the rest of the application to improve the maintainability of the application and ease complexity related to business-specific needs.

High update rate: Android applications have a high update rate because of bug fixes and the frequent addition of new features based on the changing business requirement and user needs. That makes the software development life cycles of the Android applications quite active [3]. For that reason, Android applications should be developed so that the addition of new features and fixing bugs can be done smoothly. Thus, problems can be solved in a time and cost-efficient manner. In that way, decreasing maintenance costs, shortening release times, and improving developer efficiency while developing Android applications could be possible.

Growing Codebases and Fast-changing Development Teams: Android applications' gets harder to maintain as the codebase and the development team grows or changes. The codebase of an Android application has to be in an orderly fashion that enables developers to read and understand the app's purpose quickly. Also, any time a new developer joins the team, the time required to onboard a new developer to the codebase is directly related to the way that the Android applications were developed.

Over the last decade, a couple of different ideas have been in place to resolve these issues in the context of Android application development, and similar views with the

same purpose are continually evolving. However, the primary purpose of all these ideas is the same. All these ideas and methodologies, whether in the form of software architecture, design patterns, or coding conventions, aim to improve the "maintainability" of the Android applications.

In the context of Mooncascade's Android team, from the maintainability point of view, the challenges explained above become even more critical since the company provides services in the forms of software development and consultancy, and the reason for this is how Mooncascade works. In other means, the team is divided into sub-teams, and these sub-teams work on different projects for different clients from different domains. Over time members of the sub-teams can change, a project can be extended, or maintenance might be needed. In this situation, the extension or maintenance of a project might need to be done with different developers. That is the point where maintainability emerges as a critical non-functional requirement. High maintainability means better code readability and understandability, less onboarding time for a new developer, easily extendable and changeable code. If projects are developed with high maintainability, such a process can be managed more time and cost-efficiently.

The team had internalized some set of tools, techniques, and technologies over time to that aim. Although these tools, techniques, and technologies used by Mooncascade's Android team are used by the Android community worldwide, the impact and benefits are empirically unknown from the software engineering perspective. Knowing the benefits and effects of these tools, techniques, and technologies used by the team in software maintainability is essential to understand how useful and practical they are. Thus, to identify what these tools, techniques, and technologies used by Mooncascade's Android team are and what benefits and impacts they bring, this study will answer the following research questions.

- **RQ1:** What are the metrics and/or methods for measuring maintainability in the context of Android application development?
- **RQ2:** What are the methods, techniques, tools, and technologies used by Mooncascade's Android team to develop quality Android applications?
- **RQ3:** How efficient and impactful are the methods, techniques, tools, and technologies used by Mooncascade's Android when developing Android applications, in terms of increasing software maintainability?

1.2 Scope and Goal

This study addresses the practices used by Mooncascade's Android team to resolve the challenges mentioned in the previous section. To this end, the study aims to present

comprehensive and up-to-date resources used by Mooncascade's Android team, including the tools, libraries, and techniques and how those are used to achieve the goal. The study will identify, understand, and share the practices and technologies used by Mooncascade's Android team. Moreover, it will present the determined practices in the forms of code samples and instructions. In this way, the study aims to facilitate resolving the challenges faced when developing state-of-the-art Android applications, which are also mentioned in the introduction section, by providing advanced techniques for developers and researchers.

Moreover, as a part of this study, interviews will be conducted amongst the Android developers of Mooncascade's Android team to evaluate the impact of the researched practices and technologies. Researched methods will also be compared to the data collected through an Android developer survey conducted amongst the Android community to support the validity and up-to-dateness. Lastly, the identified and studied practices will be evaluated from the maintainability point of view by using software quality metrics.

1.3 Contributions

This study's main contribution is providing comprehensive information regarding the development of large scale and enterprise Android applications through the experience of a proficient software development company, Mooncascade. Since the study topic is highly coupled to industry and industry trends, the author contributes to identifying and understanding the practices used by one of the top software development companies in the region and their impact and bringing those practices into the academy through this study. With the help of the information that this study offers, developers and researchers will know the fundamental principles and technologies required to develop state-of-the-art Android apps that have increased maintainability and quality. Thus solving the difficulties explained in the introduction section can be facilitated.

Examining white and gray literature has shown that there is no similar case study regarding Android application development. Still, some studies focus on maintainability and the quality of Android applications and solutions for challenges mentioned earlier. However, the examination has also shown that such studies lack detail and up-to-dateness. This study's valuable set of information might help fill the gap of outdatedness and lack of detail between the industry and academia in developing Android applications. Unlike most similar studies, in this study, the investigated principles and technologies will be gathered from the industry's real-life Android application development best practices. The study will include detailed information about the implementation of these principles and technologies. Apart from the main contribution, the evaluation results of presented methodologies and technologies will provide empirical data, both from the maintainability point of view and the Android developer's perspective. Lastly, the

Android developer survey conducted as a part of this study can provide insights from the latest technology trends in the Android community.

1.4 Thesis Outline

The rest of this study is structured as follows:

2. Background:

This section starts with some basic information about the Android environment and the nature of the Android applications. With this information, it aims to facilitate understanding of how Android's nature affects the maintainability and complexity of Android apps when developing Android applications.

Later, the section continues with some programming and software engineering fundamentals. First, it describes "maintainability" from the software engineering point of view. Later, it explains why these terms are essential for software engineering and then mainly in Android application development. Besides, the information regarding the issues caused by the lack of maintainability will be discussed. After that, the section explains what software architecture is and what relation it has with software maintainability. Besides, in general, software architecture in Android will be covered, and an overview of some popular solutions from the industry will be shared. Lastly, an overview of the white and gray literature review will be conducted as a part of this study.

3. Research Methodology

In this section, the metrics used while evaluating the tools, techniques, and technologies used by Mooncascade's Android team in terms of maintainability will be explained. Besides, the contents of the survey and interview to be made for the same purpose will be discussed in detail. Thus, the first research question will be answered in this section.

4. Mooncascade Case Study

The identified practices used by Mooncascade's Android team to tackle the problems mentioned in the introduction will be shared in this section. Also, extensive information on technologies and third-party libraries used by Mooncascade's Android team to achieve the goal of developing state-of-the-art Android applications with high maintainability will be provided. The practices, technologies, and techniques used by Mooncascade's Android team to accomplish the goal of developing state-of-the-art Android applications will be shared in the form of instructions and coding examples. The second research question will be answered in this section.

5. Evaluation

The impact of the practices that are used by Mooncascade's Android team on the maintainability of Android applications will be evaluated in this section. The evaluation will be both in qualitative and quantitative form. To fulfill the evaluation in the qualitative form, surveys will be conducted amongst Mooncascade's Android team members. The quantitative evaluation will be fulfilled by using software maintainability metrics. Lastly, in order to identify the accuracy of the practices used by Mooncascade's Android team, a developer survey will be conducted amongst the Android community, and the results will be shared. The third research question will be answered in this section.

6. Discussion

This section will present the discussion and interpretation of answers to the research questions and evaluation results gathered in the previous section. Outcomes of the evaluations will be shared. The pros and cons of the researched practices in this study and their impact on Android application development will be interpreted from the maintainability point of view. Lastly, the limitations and restrictions of the study will be mentioned.

7. Conclusion

The study's epitome, along with the final thoughts and comments, will be presented in this section. Also, future research opportunities will be discussed.

2 Background

This section starts with brief insights from the Android environment and Android application components in order to provide information about the nature of the Android environment and Android applications. With the light of this information, the study aims to facilitate the understanding of difficulties arising from Android's nature. The section continues by describing some fundamental software engineering concepts, including maintainability and software architecture, which the solutions when solving difficulties encountered when developing Android applications are mostly influenced. Besides, the importance of these concepts in software engineering and particularly in Android application development is explained. The section concludes by presenting the white and gray literature review results. Through all this information, it is aimed to facilitate understanding of the foundation sources of the main problems that developers often encounter in Android application development processes, which were discussed in more detail in the "Problem Statement" section. It is also another goal for readers to be familiar with the basic principles of software engineering, which is the starting point of possible methods used in solving these problems.

2.1 Android OS

Android is an open-source operating system for mobile devices. The Android project/operating system was initially created by the Open Handset Alliance which includes organizations from various industries such as Google, Vodafone, T-Mobile, LG, Huawei, Asus, Acer, and eBay to give some examples [4]. To be more specific, Android is an open-source software stack made for a varied range of mobile devices with different structure parameters. The main goal of the Android project is to provide an open software platform accessible for a variety of stakeholders such as developers, engineers, carriers, and device manufacturers to turn their innovative and imaginative ideas into successful real-world products that improve the mobile experience for the end-users. Today, numerous organizations from Open Handset Alliance and also other organizations are supporting and investing in Android and the project is led by Google. Android is designed in a distributed way to avoid the issue of the central point of failure. In another means, different industry players confine or control the advancements of another. As a result, a production-quality consumer product comes along with open source code that is ready for customization [5].

The platform architecture of Android consists of 6 major layers. Each layer has its own responsibility and handles a different area of the Android operating system. The following figure demonstrates the layers in a way that they are ordered from the highest level of abstraction from the top to the bottom.

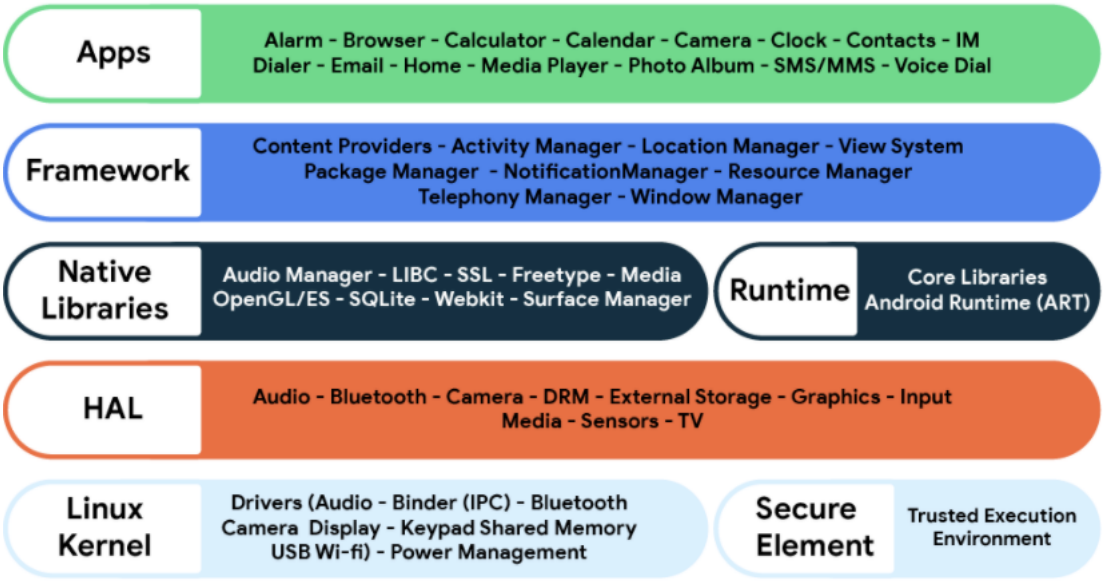

Figure 1. Android platform architecture [5]

The section below gives a brief description of each major layer of the Android platform architecture mentioned in Fig. 1:

2.1.1 Applications

The application layer is the top layer of the Android platform architecture. The Android operating system provides a decent number of built-in applications for fundamental user needs such as contacts, calling, web browsing, email, SMS, calendars, and more [6]. These built-in applications are developed by the Android team. In addition to these built-in applications, there are also third-party applications that are developed by third-party providers. All these applications are installed in the application layer. Android provides the flexibility for developers to write applications that can replace any built-in application provided by the Android operating system. This study focuses on the best practices to develop the maintainable and testable state-of-the-art Android applications that will be installed in the application layer.

2.1.2 Java API Framework

The Android operating system's APIs written in Java programming language are provided through this layer. These APIs are the core building blocks for the developers in order to develop Android applications. A couple of noticeable key core building blocks include

the following [6]:

- View System: It provides an extensible set of views for creating user interfaces and user experience.
- Resource Manager: Providing tools for accessing and managing non-code embedded application resources such as strings, colors, values, layouts, images, etc.
- Activity Manager: It provides the tools for managing the application lifecycle and navigation back stack.
- Content Providers: It provides tools for enabling data sharing between different applications.
- Notification Manager: It provides tools for developers to add the ability of notifications and alerts for Android applications.
- Location Manager: Provides tools for developers to manage location-related data and location updates.

The practices that will be covered in this study mainly occur on this layer because the code is developed and structured in the Java API Framework layer.

2.1.3 Native Libraries

This layer includes a variety of C/C++ core libraries and Java libraries. The purpose of this layer is to provide support for core features. Important system components of Android, such as HAL (Hardware Abstraction Layer) and ART (Android Runtime) depend heavily on the native libraries that are written on C/C++ programming languages [6]. Some of the important native libraries are Camera, Media, OpenGL, SQLite, WebKit, SurfaceManager, etc.

2.1.4 Android Runtime

Android Runtime(ART) is a runtime environment for applications that run on the Android operating system. With the launch of Android Version 5.0 or in other words, API level 21, the Dalvik virtual machine was replaced with Android Runtime. Since then, every application started running on its own process along with its own instance of ART [6].

2.1.5 Hardware Abstraction Layer

Hardware Abstraction Layer (HAL) is responsible for providing the device hardware features to the Java API Framework layer through interfaces [6]. In other words, it enables the communication between the device hardware and framework. The hardware includes important device features such as Bluetooth, camera, and sensors.

2.1.6 Linux Kernel

The very core of the Android platform is the Linux Kernel and the base infrastructure of the Android heavily depends on the Linux. Using Linux comes with some advantages. Firstly, using a well-known kernel enables device providers to work on a platform that they already recognize, and second Android benefits from the Linux system and kernel security [6]. This layer also provides all the hardware drivers for camera, display, etc., and handles battery management and other system-related properties.

2.2 Fundamentals of Android Applications

This study mainly focuses on the way that Android applications are developed. The motivation of this study is to develop state-of-the-art Android applications with high maintainability. However, before discussing these technologies and methodologies it is essential to know the fundamental components that create an Android application.

Ever since the Android operating system has started running on mobile devices, Android applications are being developed. As of the first quarter of 2020, there are more than two and a half million applications in the Google Play Store [7]. Since the launch of the first mobile device that works with Android, the Android operating system has improved, and the way the Android applications are developed changed a lot, but some fundamental components for developing Android applications have more or less stayed the same. Each of these fundamental components was developed for a specific purpose by the creators of the Android Software Development Kit (Android SDK). Knowing these components and their responsibilities is necessary to understand the problem that this study is trying to solve. Because as it was already stated in the introduction section, one of the reasons for the complications that arise when developing Android applications is the nature of the fundamental Android components. So, in this section, some brief information will be given about these fundamental Android components. For more information and technical details regarding these components, it is recommended to read official Android documentation.

Java, Kotlin, and C++ programming languages can be used for developing native Android applications [8]. There are also other ways of developing Android applications.

But as it was already mentioned in the introduction section, this study only focuses on native Android application development. Native Android development means the creation of Android applications that run on Android-powered devices by using the Android Software Development Kit. When developing Android applications in a native way, in addition to the programming side, Android applications are supported by different types of resources such as XML layout files, XML resources, images, data files, etc. The detailed examination of these resources is not within the scope of this study. However, knowing that Android applications do not only consist of code might be useful to see the bigger picture of an Android application. Though, for the purpose of understanding the problem that this study tries to resolve, it is essential to have a basic understanding of the fundamental Android components. Consequently, knowing the nature of an Android application and its components is the first step for solving the maintainability issues of the Android applications and then there come the best practices and latest technologies of Android application development and how to apply them into the Android application development processes. These fundamental components can be listed as [8]:

- Activities
- Services
- Broadcast receivers
- Content providers

The remaining of this section will introduce fundamental Android components and the brief information regarding these components to help to understand the problem that is stated in the study. Since the impact of activities is bigger when it comes to the maintainability of Android applications, the information to be provided regarding this component is slightly more detailed when compared to the information regarding the rest of the components.

2.2.1 Activity

In the Android environment, the term "activity" refers to the interaction entry point of Android applications for the end-user. Alternatively stated, an activity is a screen with a user interface. It would not be wrong to say that activity is the main component for building an Android application. Activities are represented by the "Activity" class of the Android Software Development Kit. Each activity of an Android application is implemented as a subclass of the Activity class [8].

An Android application might have single or more activities. Oftentimes an Android application consists of a set of activities. An activity can start and finish another activity

which means activities are also responsible for the navigation within the application. An activity can even start another Android application from the device. Activities within an app provide a strong user experience together. Since the activities of an Android application can affect each other and each activity has its own lifecycle, this situation has an important impact on the maintainability of the Android applications. Thus activity lifecycle should definitely be taken into account when developing Android applications. During the lifetime of activities, some actions should be taken and some methods should be called depending on the situation. Some of the important ones of these situations are covered in the upcoming sections but before discussing these, it is important to understand the activity lifecycle. The figure below is an illustration of the activity lifecycle and it presents the activity lifecycle methods.

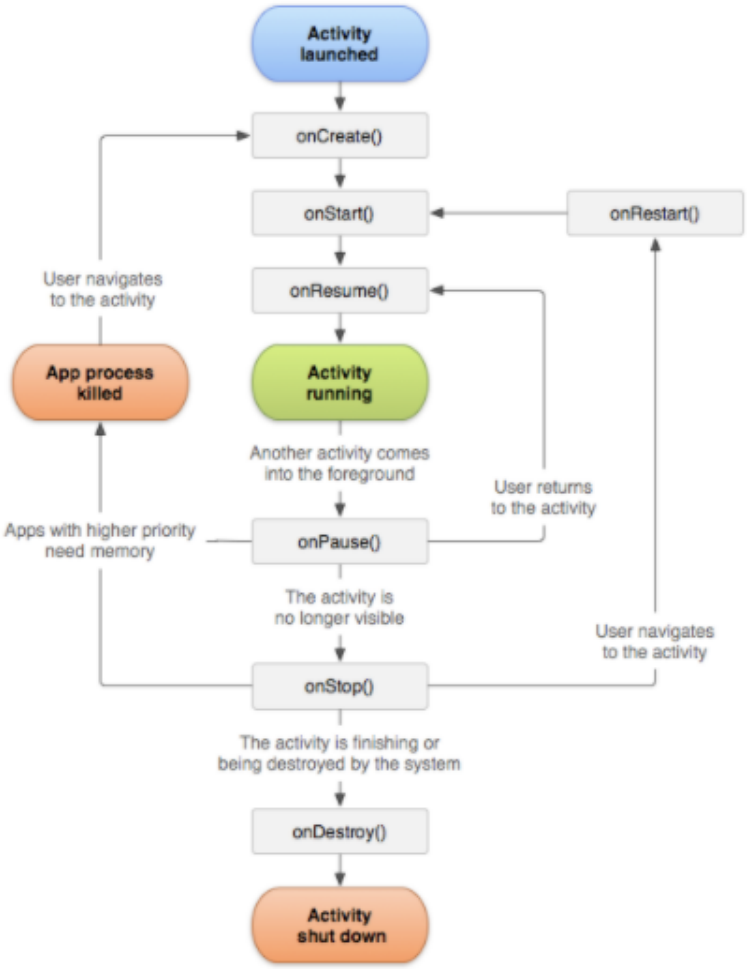

Figure 2. Android activity lifecycle [8]

Users navigate between the different Android applications and/or navigate within screens of an Android application. During this navigation instances of activities are created, destroyed, or put into different states of the activity lifecycle. The Activity class from the Android SDK offers several different callback methods that enable the activity to recognize the state changes. Those methods are called activity lifecycle callback methods. With the help of these callback methods, developers can decide how activities behave during state changes. Proper usage of the activity lifecycle callback methods helps developers to prevent situations such as using valuable Android system resources when users do not really need them, losing the users' last state when they leave an application, crashing when switching between different applications and losing the last state during device orientation changes.

Improved knowledge of activity lifecycle callback methods is not only helpful for maintaining system resources, organizing the activity behaviors during the state changes, and preventing undesirable situations but also useful when it comes to integrating non-UI related components of an Android application to the activities. It is essential for using activities and these other components together in harmony with high efficiency. Good knowledge of these callback methods enables developers to be aware of when to add/release these non-UI related components to activities when building maintainable Android applications. Detailed explanations of the importance of the activity lifecycle methods and their functionality can be found in the Android developer guidelines [8].

2.2.2 Fragment

It is previously mentioned that activity represents a screen with a committed user interface. Nevertheless, an activity might also include one or more fragments. In the Android world, the term "fragment" is used to picture a behavior or a part of a user interface in an activity. Although fragments are not one of the main Android application components that were previously mentioned in this study, they are worth discussing briefly due to the fact that fragments have their own lifecycle and they are highly connected to the activities in that sense.

A fragment can be considered as a modular part of an activity. Fragments can be combined in an activity to build multi-window user interfaces. Fragments have their own lifecycle and input events. Unlike activities, fragments are reusable and reuse of a fragment in multiple activities is possible. Fragments cannot exist on their own. Every fragment is hosted by an activity. That makes the lifecycle of a fragment directly related to the lifecycle of the host activity that the fragment was created in. Fragments are represented by the "Fragment" class of the Android Software Development Kit. In order to create a fragment, a class must inherit the Fragment class or existing subclasses of the Fragment class. It was already indicated that, though fragments have a lifecycle that is

tightly coupled with their host activity lifecycle, they have their own lifecycle which is different than the activities' lifecycle. Thus fragments have their own lifecycle callback methods. The study already aforementioned that why having good knowledge regarding activity lifecycle callback methods is important. The importance of the fragment lifecycle callback methods is no different as well. Proper understanding of fragment lifecycle callback methods helps developers to know when to add/release these non-UI related components to fragments when building maintainable Android applications based on fragments. The figure below is an illustration of the fragment lifecycle and it presents the fragment lifecycle methods.

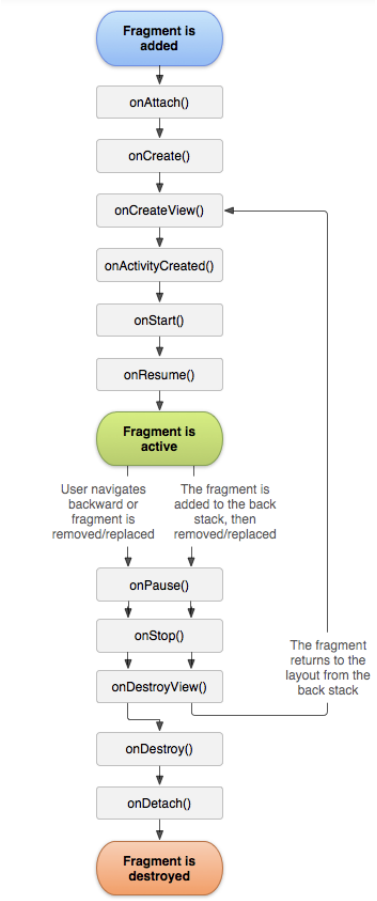

Figure 3. Android fragment lifecycle [8]

In the same way as activity lifecycle, the fragment lifecycle exists in three states as well and these states are "resumed", "paused", and "stopped". Transitions between these states are managed by the system through the callback methods presented in the figure above. Detailed explanations of the importance of the fragment lifecycle methods and their functionality can be found in the Android developer guidelines [8].

2.2.3 Service

In the Android world, the term "service" refers to a component that runs in the background in order to perform time-consuming and long-running processes or to perform remote processes. In other words, a service is an entry point for Android applications to keep the applications running in the background for any reason. A service does not have a user interface. Other Android components such as activities, fragments can start a service. Once a service is started, it continues to its long-run even if the user starts another application. Android services can be used to perform background operations such as network operations, content provider interaction, I/O processes, and playing music. In Android, a service is represented by the "Service" class and every service must be implemented as a subclass of the Service class that the Android SDK provides [8].

2.2.4 Content Providers

In the Android world, the term "content provider" refers to the component that is designed to manage a mutual application data set that can be stored via a file system or a local database (e.g. SQLite) or any other kind of persistent storage. Content providers define, manage, and supply inter-application data sets. Android applications can provide content providers for other Android applications and through the content providers, any other Android application that has the necessary permissions can query the content provider to read and write data within its permissions. From the Android system perspective, a content provider can be considered as an entry point into an Android application in order to issue named data sets identified by a URI scheme. A solid example to a content provider can be given as the content provider that the Android system provides for the purpose of managing the contact information of the user between multiple apps [8].

2.2.5 Broadcast Receivers

The term "broadcast receiver" refers to the Android system component that enables the system to distribute events that happen outside of the normal application flow to the Android applications. Just like the other main Android application components that were mentioned previously, broadcast receivers are also entry points to the Android applications. As a result of that, the Android system can deliver broadcasts to Android applications regardless if the application is running or not. Broadcast in Android tends to originate from the Android system itself. Low battery notification, captured screen notification, and the screen on/of indicators can be given as examples to the Android system broadcasts. However, Android applications can also initiate broadcasts. Broadcast receivers do not involve displaying a user interface but they have the ability to create notifications in the status bar in order to alert users. Android Software Development Kit

provides the "BroadcastReceiver" class and each broadcast receiver must be implemented as a subclass of this provided class in Android applications [8].

2.3 Maintainability

"Good programmers write code that humans can understand." - Martin Fowler [9]

In ancient times, when computers were big, heavy, and slow, programmers were limited to use low-level programming languages that are working close to computer CPUs. These were imperative programming languages, and the programs written in these programming languages were following the procedural programming paradigm. Although that approach worked fine, the biggest problem was that these programming languages were designed to be understood by computers, not humans. The main reason for this situation was that, back in that time, computers lacked proper hardware, resources, and speed. Consequently, the priorities back then were different. The computer programs had to be fast and less memory consuming. However, this situation has changed in the current day. Even a low-quality mobile device is much stronger and smarter than the computers that people were using a couple of decades ago, and software systems became complex. Although this change brought a positive impact on the end-user side as it also brought more functionality and ease, the impact it brought the software development side is complexity. Especially when developing large-enterprise software products, ignoring that fact and not considering how to overcome this complexity may cause significant failures. In this context, new programming languages and paradigms were born, the priorities have altered, and this new reality brought different challenges and new quality standards to software development [10].

When the priorities of modern software development are analyzed today, the maintainability of software systems emerges as one of the most critical priorities, perhaps even the most important. According to the IEEE Standard Glossary of Software Engineering Terminology, the term "maintainability" is the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed [11]. In the context of software engineering, maintainability means how well a software system is understandable, repairable, and extendable. Maintenance is one of the most important parts of the software development life cycle because the time spent on maintaining software systems requires more time and resources than the rest of the process. The relative expense for maintaining software and dealing with its development speaks to over 90% of its absolute expense [12]. The level of maintainability that a software system depends on several different factors. Overall, a software system can be considered maintainable if it is simple to grasp how it works and what it does, and making changes such as adding new features and fixing bugs is easy. Besides, maintainability is directly related to well-known software engineering concepts such as

coupling, complexity and cohesion. Knowing the relationship between these concepts and maintainability is also essential in measuring maintainability, which is one of the topics of this study. Previous studies have proved this relationship. The inverse ratio between complexity and maintainability was mentioned in Saifan and Rabadi (2017) on measuring maintainability in Android applications [13]. Also, the inverse relationship between coupling and maintainability and the correlation between cohesion and maintainability are discussed in detail in Barak et al., (2012) on maintainability metrics in open-source software [14].

Considering the life cycle of software systems might help to understand the importance of maintainability. Software systems are born, they live, they change, and eventually, they die. However, their lifetime is generally long, and during their lifetime, new features are added, some features are removed, bugs are fixed, and often their development team changes. Usually, there is always a time gap between these changes. In other words, developers might need to make a change to the software system, which they worked in weeks or months before. In such cases, developers should be able to understand the systems easily even months after. Besides, changes to the code base should be able to be done with ease without breaking the other parts of the software system. Also, when a new developer joins the development team, onboarding should be smooth, and the new developer should be able to understand the purpose of the software system easily. Ignoring all these might cause companies a significant amount of time and money. That is what makes maintainability that important. Developing maintainable software systems is the way to tackle such issues.

The importance of maintainability for software systems can also easily be seen when looking at its role in the software development lifecycle and its effect on software development costs. The maintenance period for a software system starts as soon as the system is developed. Thus, maintainability becomes a vital aspect for applying new customer needs, adding/removing new features, adapting to the environmental changes [15]. The time that has to be spent on the maintenance of complex software products is comparatively more extended than the rest of the software development lifecycle processes. Reports indicate that the amount of effort spent on software maintenance is between 65% and 75% of the total amount of effort [16]. Also, another report points out that maintenance cost is 75% of the total project cost, and the cost for maintaining source code is ten times bigger than developing the source code [17]. In his famous book "Clean Code", Robert C. Martin explains how a top-rated company in the late 80s was wiped out from the business due to the lack of maintainability and poorly managed code organization. When the release cycles of their prominent product extended, due to the unorganized code base of their product, they were not able to fix bugs, prevent crashes, and add new features. Eventually, they had to withdraw their promising product from the market and went out of the business. Lousy code and consequently, maintainability was the reason for this company to go out of the business [18]. This real-life example clearly

shows how vital maintainability is to software systems and what fatal consequences it can cause if ignored.

The importance of the maintainability for software systems is evident and this situation is no different for Android Applications. In fact, the importance of maintainability for Android applications is even higher since Android applications have a very active software development life cycle. Given that the growing user demands and business needs making the Android applications more and more complex and Android applications having frequent update rates, it is not hard to see how important the maintainability is for Android application development. This situation is one of the main sources of motivation for this study.

In addition to that, in the context of Android, when the main challenges mentioned in this study are evaluated together, the importance of maintainability as a non-functional requirement becomes even more evident for Android application development because the high level of maintainability is the way to overcome the challenges and complexities mentioned in this study while developing Android applications. In consequence, the question is how to achieve the goal of developing Android applications with high maintainability. From the software development point of view, the Android platform does not have strict rules on how the applications are developed. Developing maintainable applications is not an obligation. However, developing Android applications with high maintainability is a need to solve the difficulties mentioned in this study in a timely and cost-efficient manner, facilitate the development processes for the Android developers, and increase the quality of the Android applications. The methods to be followed and the technologies to be used in Android applications for meeting these requirements constantly evolve, and the topic is still controversial among the Android community. Different solutions have been proposed and tried since the birth of the platform. However, the unchanged reality is that the most important criteria for building reliable software systems in a timely and cost-efficient manner is maintainability and of course, this reality is not different for Android applications.

2.4 SOLID Principles

In the previous section, the importance of maintainability and maintenance of software systems were explained. The steps to be taken to increase the maintainability of the software systems can be taken at the design stage. Thus maintenance costs of software systems can be minimized. To that aim applying some techniques and principles is essential when developing software systems. The SOLID Principles are very efficient when it comes to solving such design issues and increasing maintainability of software systems. SOLID stands for five principles, namely Single responsibilities principle, Open close principle, Liskov substitution principle, Interface segregation principle, and

Dependency inversion principle. Application of SOLID principles when developing software systems via object-oriented programming facilitates improving critical factors such as maintainability, extendability, readability, and reducing code complexity and tight coupling, increasing cohesion [19]. Recalling from the previous sections, factors such as maintainability, extendability, readability, increasing cohesion and reducing code complexity and tight coupling are closely related to the challenges that are faced when developing Android applications. Thus it is wise to use programming techniques that can ensure the application of SOLID principles when developing Android applications. A detailed description of the SOLID principles is beyond the scope of this study. However, briefly summarizing each principle as below will be enough to remind the principles given that the readers of this study are already aware of them. Following are the brief descriptions of each principle [19]:

- **The Single Responsibility Principle:** A class should have only one reason to chance.
- **The Open/Close Principle:** A module should be open for extension but closed for modification
- **The Liskov Substitution Principle:** Subclasses should be substitutable for their base classes.
- **The Interface Segregation Principles:** Many client specific interfaces are better than one general purpose interface
- **The Dependency Inversion Principle:** Depend upon Abstractions. Do not depend upon concretions

Not following SOLID principles may lead to serious maintainability problems in the software development lifecycle, such as tight coupling, code duplication, and bug fixing. Application of SOLID design principles when developing software systems helps to achieve essential quality factors such as understandability, flexibility, maintainability, and testability [19].

2.5 Separation of Concerns

Today, the complexity of software applications is quite high, and it still is increasing. Today's software systems have many complex concerns such as persistence, real-time constraints, concurrency, visualization, location control [20]. Software engineering, first of all, aims to increase software quality, lower the expenses of software production, and assist maintenance and development. In achieving these aims, software engineers are

continuously on the lookout for developing technologies and approaches that increase maintainability, decrease software complexity, enhance understandability, support reuse, and boost development. That is where "Separation of Concerns" (SoC) emerges as a solution.

In the context of software engineering, SoC is a software design principle for separating a software system into discrete modules that each module addresses a single concern. A good application of SoC to a software system provides benefits such as increasing maintainability, reducing complexity. The borders for different concerns might differ from a software system to another. Concerns depend on the requirements of a software system and the forms of decomposition and composition [21].

In the context of Android, the importance of SoC is even more apparent. Android applications have different concerns with clearly drawn borders. Such concerns can be named as visualization and presentation of data, business logic, networking, persistence, location services. Depending on the requirements of an Android application, there might be other concerns as well. In light of the information above, it is vital to identify these concerns before Android applications are developed, and during the development phase, to use techniques that can apply SoC well.

2.6 Software Architecture

"Good architecture makes the system easy to understand, easy to develop, easy to maintain, and easy to deploy. The ultimate goal is to minimize the lifetime cost of the system and to maximize programmer productivity." - Robert C. Martin [22]

The previous section explained that software systems are not static environments. They grow, change, and get updated based on the new requirements. During this change, new additions are made in forms of components, methods, modules. As the system grows, the interactions between these different system elements also get more complicated and all together lead to complexity in software systems. In order to be able to develop reliable software systems that can overcome this complexity, programmers must implement software systems in a generalized fashion. Software systems that are written in such fashion can be considered reliably usable, maintainable, testable, and extendable [23].

At that stage, the question is, what would be the so-called generalized fashion from the software engineering perspective. A couple of decades ago, software engineers and researchers started paying more attention to this topic and started studies to find an answer to this question. These studies were intended to find solutions for the problems emerging while developing large scale software systems. The focus area of these studies started as software design and eventually evolved into software architecture [24]. The answer to the above question today is software architecture. Martin Fowler defines the

meaning of the architecture in the software industry as "the shared understanding that the expert developers have of the system design" [25]. Technically speaking, software architecture is "the set of significant decisions about the organization of a software system, the selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these elements into progressively larger subsystems, and the architectural style that guides this organization – these elements and their interfaces, their collaborations, and their composition" [26].

Good architecture facilitates the satisfaction of requirements for software systems as well as providing better performance and more reliability. Today the impact of software architecture on the success of software systems is quite high. Selecting the right architecture for a software system is a critical success factor for system design and development. As Brian Foote quotes, "if you think good architecture is expensive, try bad architecture"[27]. Also, software architecture has a central role between software implementation and requirements. The role of software architecture in software development can be elaborated under the six major aspects. Those aspects can be listed as follows [28]:

- **Understanding:** Good architectural design simplifies the understanding of systems, in the context of what the system does and how it does as well as facilitating the understanding of the systems' high-level design. Thus good architecture improves the readability of software systems.
- **Reuse:** Good software architecture facilitates and increases the code reuse between the components.
- **Construction:** Software architecture provides a good outline of the software system in the form of layers and abstractions. The outline describes the components of the system and dependencies between those components.
- **Evolution:** Software architecture can point out the dimensions which a software system might evolve to. Therefore software architecture helps software teams to understand the consequences of changes more accurately. Moreover, software architecture describes how the concerns of the software systems are separated and it determines the boundaries of interactions between those concerns.
- **Analysis:** A well-applied software architecture enables analyzing the system consistency, compliance with restrictions imposed by the architecture, and compliance with the quality attributes.
- **Management:** Assessment of software architecture facilitates understanding of risks, requirements, and implementation strategies.

When contemplating the importance of software architecture for Android application development, of course, there is no difference. When the fast update rate of Android applications, frequent requirement changes of Android projects, and the complex issues and restrictions that arise from the nature of Android are taken into consideration, the significance of software architecture selection when developing Android applications becomes even more prominent. However, this area has been problematic for Android application development and Android developers from the beginning.

During the early days of Android development, Android developers tended to use "god activities". God activities were activities where developers place all kinds of code including user interface, presentation logic, business logic, and so on. This unstructured way of a software development approach that does not follow any architectural or design pattern is called "anti-pattern". Back then, Android applications were relatively smaller and less complex. Ever since the Android applications started becoming more and more complex these god activities became a black hole for Android developers because they were hard to read, understand, test, and maintain. Android developers were quick to recognize these problems and needs and had no trouble finding solutions on how to develop Android applications in a more organized way. In order to solve this god activity and such an anti-pattern problem and improve the maintainability of Android applications, Android developers started applying well-known and widely used architectural and design patterns in GUI-heavy applications to Android application development [29]. With improvements and advancements in hardware and software and growing demand for the user and business needs, the requirement for developing organized and maintainable Android applications has had a great extent [30].

Today, though the use of architectural patterns and design patterns in Android application development does not seem to be a requirement, it is actually of a de-facto must. Developing an Android application without applying an architectural pattern would be a bad decision because complex applications that do not follow any architectural pattern are expected to end up with serious maintainability issues. Also, surviving in such a competitive market is heavily dependent on developing a well-architected Android application that has a high level of maintainability. While developing Android applications, the expectation in terms of software architecture is to apply the SOLID principles and separation of concerns in a healthy way and to have high maintainability. On the other hand, there is no fixed solution for that problem that Android demands from developers to meet its applications and there are a variety of options. So, the right way to architect Android apps still remains in discussion with clashing standpoints which are generally affected by technological hypes [31].

Importance of the software architecture in terms of software systems and especially Android applications was explained above. At this point, it will be of utmost importance to emphasize the relationship between architectural choice and maintainability, as the

main subject of the study is the improvement of the maintainability of Android applications. Can it be claimed that maintainability is the most crucial quality requirement when developing Android applications when it comes to architectural selection? With many options available for architecting Android applications, what is the top quality requirement that an Android application architecture should provide in order to overcome the complexities? As this study has emphasized continuously since the beginning, the answer to these questions is maintainability. A related study had conducted a questionnaire between Android practitioners and researched other related academic papers and results had revealed that the top quality requirement for architecting Android applications is maintainability [31].

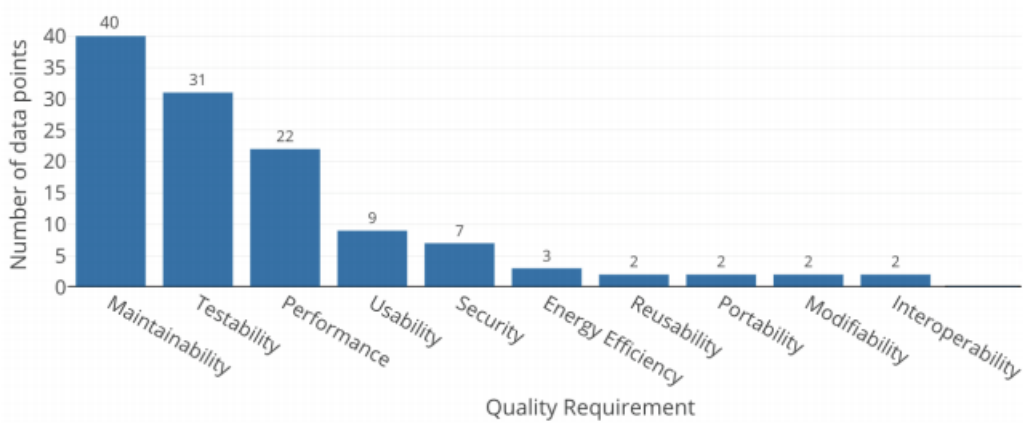

Figure 4. Quality requirement rankings for architecting Android apps [31]

It is proven that maintainability and selection of architectural patterns when developing Android applications are directly in correlation. In addition, it can be said that other quality requirements presented in the figure above, apart from maintainability, directly affect maintainability itself. In particular, quality requirements such as modifiability, testability and reusability will increase as the maintainability increases, or vice versa. As a consequence, the impact of the architectural pattern on the maintainability of Android applications is one of the top aspects that should be considered when making the decision of the architectural pattern before starting the development of a new Android application.

2.7 Literature Review

In this section, the outcomes obtained from examining the academic studies on Android application development are presented. Considering the tight relationship between the topic and industry, a Systematic Literature Review (SLR) was not adequate for finding relevant resources of data for the study. Hence, a Multivocal Literature Review (MLR)

was conducted. As a type of SLR, MLR is collecting grey literature as well alongside formal literature [32]. MLR considers resources like blogs, white papers, articles, academic literature and allows gathering information from academics, developers, practitioners, and independent researchers [33]. For a topic closely related to the industrial trends, adding grey literature next to the white literature as a part of the literature review was crucial. Because the most up-to-date resources are the grey literature resources in the Android environment and to compare the white literature's situation to the latest industrial trends of the Android application development, grey literature is definitely needed. Regardless of the direct relevance of the studies to maintainability, the following research query has been used to determine all the studies related to Android application development published in the last five years. Among the determined studies, those related to maintainability have been examined in more detail.

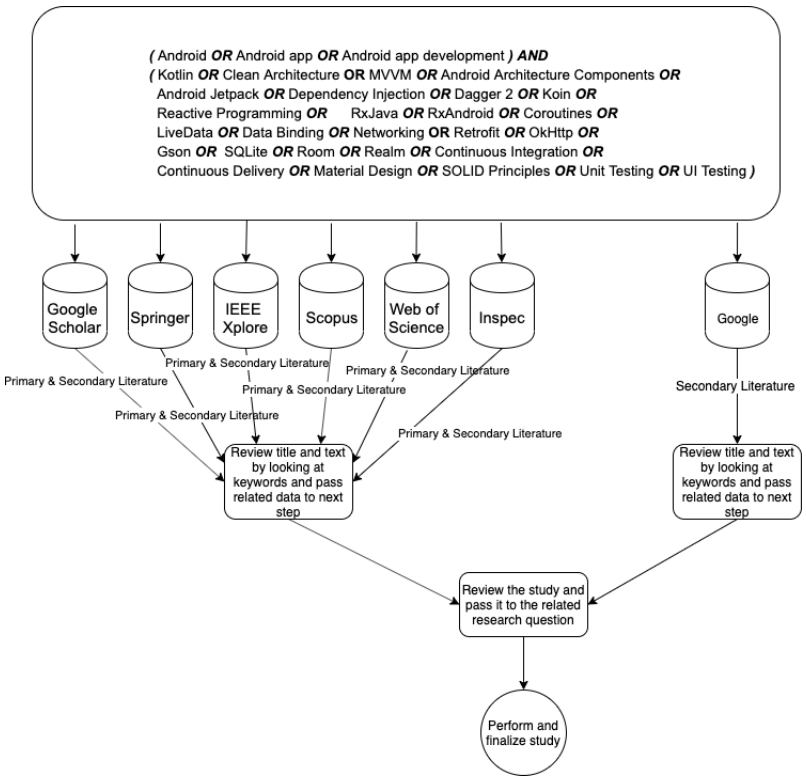

Figure 5. Search process visualization

As a result of the executed research query above, more than forty seven papers were found and reviewed. Grey literature results are not included in these numbers. The found and reviewed grey literature results were only used to compare the academic literature and industry's situation and interpret the situation. Also, the reviewed grey literature was cited in this study. For the purpose of obtaining more successful findings, the search

query was separated into six parts, each of them individually focusing on a single topic. The literature found as a result of this query was read thoroughly. Finally, inclusion and exclusion criteria were applied to the primary studies to have suitable literature only. As soon as the early research outcomes were collected, chasing the inclusion and exclusion criteria shown in Figure 6, the unrelated material was filtered out. Also, the studies conducted before 2015 are excluded from the query results to prevent any possible up-to-dateness issues.

Inclusion Criteria	Exclusion Criteria
Studies that discuss Android apps	Studies not published in English
Studies that discuss Android Architecture	Studies published before 2015
Studies that discuss maintainability of the Android applications	Studies not focusing on native Android applications
Studies that discuss Dependency Injection for Android Applications	Studies that containing outdated practices/information about Android
Studies that discuss Reactive Programming for Android Applications	Unavailable studies
Studies that discuss 3rd party libraries	Studies with commercial purposes

Figure 6. Inclusion and exclusion criteria

These more than forty academic studies examined on Android application development can be grouped according to their topic titles as follows.

- 4 studies bachelor theses regarding Android application development.
- 5 papers were related to 3rd party Android libraries.
- 5 papers were related to Kotlin programming language usage for Android.
- 5 papers were related to maintainability of the Android applications.
- 12 papers were related to Android app architecture.
- 16 papers were related to Android application development but not directly related to the topic that this study covers.

In addition to the papers mentioned above, 15 other papers are reviewed to get a better understanding of the studies conducted regarding software maintainability, metrics that can be used to measure software maintainability, general software engineering principles

such as separation of concerns, and SOLID. These articles and books have been cited in various parts of this study.

Among these more than forty academic studies on Android application development that are reviewed as a part of this study, the studies on Android application architecture and maintainability draw attention. Studies conducted on these two subjects account for more than one-third of the total studies examined. Based on these numbers, it can be said that the Android application architecture and the importance of maintainability are recognized by the researchers working in the Android field. As stated in the 2.3 and 2.6 sections of this study, the importance of application architecture and maintainability in the context of Android application development processes is supported by the above numbers and academic studies included in these numbers. In most of these studies, it is noteworthy that comparisons of Android application architectures in terms of performance, maintainability, and testability are common. As mentioned in previous chapters, considering the impact of software architecture on maintainability and the importance of maintainability in software development processes, it is not surprising that many academic studies have focused on architecture and maintainability. Various studies on the maintainability of Android applications draw attention. For example, Hugo Källstrom conducted a study on a similar topic in which he implemented three different software architectures and evaluated the maintainability based on these architectures [30]. Also, Prabowo et al. have a research on the maintainability of Android applications which they compared MVP and anti-pattern approaches [29]. Apart from these works, the work of Verdecchia et al. (2019) on architectural choices in Android applications is a worth reading work on both architecture and maintainability topics [31]. These studies regarding the architecture and maintainability in Android application development stand out as resources worth examining. There are also some other studies that are cited in the paper, they are worth reading too. On the other hand, there are some significant issues with most academic studies regarding the topic.

First of all, it is noteworthy that academic studies in this field are inadequate. Even as a result of a comprehensive literature review that included the grey literature, around fifty studies were reached, and this number is the total number of studies on Android development. The number of studies on the maintainability of Android applications is slightly less than one-third of this total number. As explained in the previous sections, considering the importance of maintainability in Android application development, it will be seen that the number of studies in this field is insufficient.

In addition to this quantitative problem, it is seen that there are some qualitative problems among the existing academic studies as well. As a result of the research among academic studies regarding Android application development, the conclusion is that the inadequacy of the formal resources in terms of up-to-dateness constitutes a major issue. Considering the dates of the studies and the continuous and rapid development of the

Android world and the changing trends, this result is not unexpected but still points to an issue. That situation creates one of this study's motivations, which is closing this gap between industry and academia. When the literature review results are evaluated in terms of maintainability, which is the main element of this study, some significant issues are observed.

When the studies on the maintainability of Android applications are examined, we see that many studies approach maintainability in terms of software architecture. As mentioned earlier in this study, it can be said that this result is not a surprise, considering that software architecture is one of the most significant factors in the impact on maintainability. Both academic resources and information gathered from the industry indicate that, when it comes to building Android applications with high maintainability, the architectural choices are shaped around the same main architectural and design patterns. Those can be named as MVC, MVP, MVVM and Clean Architecture. However, When the existing studies are examined, most of the studies cover the basic implementation of these design patterns and comparisons between these design patterns in terms of performance and maintainability.

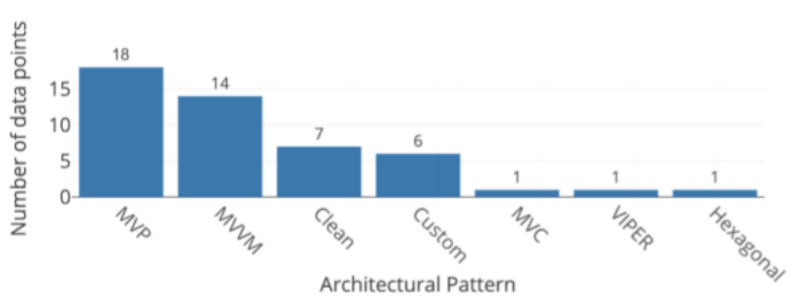

Figure 7. Well-known architectural and design patterns for Android [31]

On the other hand, considering the other factors affecting the maintainability of Android applications and some of the outdated techniques and technologies used in the studies mentioned above, it is possible to talk about the insufficiency of these studies on Android application development in the last five years. The systematic review indicates that the lack of detail and outdatedness in these studies is apparent. Another problem that strikes the eye among the academic studies reviewed is that most studies only approach the maintainability of Android applications from an architectural perspective. It is possible that other techniques and technologies can be used to increase the maintainability of Android applications besides architectural patterns. The fact that the studies do not focus on these techniques and technologies can be considered a deficiency of the academic literature regarding this topic.

In addition to the above-mentioned situation, in almost all of the examined academic studies and the reviewed work from the industry, the fact that these design patterns are presentational design patterns and they are not architectural patterns is completely ignored. In other means, the patterns derived from the MV-I concept, such as MVVM, MVP, MVC, are actually designed for how data is managed for display purposes. MV-I design patterns are intended to control the communication between the view layer and the data layer of GUI heavy applications. However, while developing small-sized Android applications, these design patterns can be effective up to a point. Nevertheless, while developing enterprise Android applications that have more sophisticated and complex business logic, it is obvious that these design patterns will be insufficient in scaling the application and solving the maintainability problems we have mentioned. Hence, the visibility of the need for higher-level architecture and other techniques when developing complex Android applications is clear. Also, considering the study topic's tight relation with the industry, knowing the technologies, techniques, and latest trends used in the industry is also essential for studies on this subject. However, when current studies are examined, it will be seen that there are problems related to these situations.

The literature review shows that the academia lacks awareness of the industry trends and comprehensive information about techniques and technologies that can be used to develop maintainable Android applications with up-to-date tools. This study aims to fill this gap in the academia by explaining and analyzing one of the top mobile application development companies' methodologies in the region.

2.8 Summary

In this section, a set of essential information regarding software engineering and Android platform information was shared, and the motivation of the study was covered. The purpose of sharing fundamental software engineering principles such as maintainability and SOLID principles is to shine a light on the background of this study's problem and facilitate understanding of the problem that this study is trying to solve. Again, with the same purpose, necessary information about the Android platform was also shared. The insights of maintainability issues in software development and the consequences of software development complexity issues were covered and interpreted in general. It was also indicated why the challenges get even more complicated when it comes to Android application development. Lastly, the conducted multivocal systematic review of the existing academic papers was shared and interpreted as part of this study.

3 Research Methodology

In the first and second sections, information was given about this study's aims, the reasons for the study's emergence, and the study's motivation sources. Besides, to better understand this study's background, objectives, and solutions, the necessary information was shared with the readers in both the Android and software development fields. As stated in the previous sections, the starting point of the study is the difficulties encountered in Android application development and a non-functional requirement named maintainability, which has an important role in solving these difficulties. As stated before, this study focuses on solving these problems in Android application development and how Mooncascade, a leading software development company in the region, approaches the issue of increasing maintainability. In this context, the approaches, techniques, and technologies used by the company's Android team will be discussed, and the level of effectiveness of these will be determined in terms of maintainability. At this point, the first question that comes to mind is how this effect can be determined or measured in terms of maintainability.

In this section, the methods to evaluate and measure how effective the techniques, technologies, and other approaches used by Mooncascade's Android team are from the maintainability perspective will be explained. These assessment and measurement methods will be both quantitative and qualitative. In this way, it was aimed to evaluate with maximum efficiency by going beyond the traditional quantitative measurement techniques. The methods to be used and the contents of these methods will be presented in detail in this section. Also, why and how these methods are chosen will be stated in this section. Thus, the first of the three research questions stated in the introduction section will be answered in this section. In the 4th section, the technologies, techniques, and approaches used by Mooncascade's Android team to increase the maintainability of Android applications and eliminate the difficulties mentioned in the previous sections will be discussed. These techniques, technologies, and approaches will be evaluated in section 5 using the measurement and evaluation techniques explained in this study. Therefore, it was thought that answering the first research question here is more appropriate before answering other research questions in the upcoming sections. The remainder of this section will provide detailed information on the quantitative and qualitative measurement techniques to be applied in this study to measure maintainability in the context of Android application development.

3.1 Qualitative Method

When other academic studies dealing with the measurement of maintainability in Android or other software systems are considered, it can be seen that many studies use scientific

and quantitative methods. The work of Verdecchia et al. in the maintainability and architecture of Android applications can be shown as a successful example of this situation [31]. Although it cannot be claimed that such quantitative measurements are wrong, it would not be wrong to say that these measures are inadequate at times. It is essential to make qualitative evaluations, and quantitative evaluations in areas where technologies are rapidly developing and trends change quickly, especially in Android application development. In this way, it may be possible to measure developers' experiences that differ in the face of rapid change and development and the effects of these experiences on the maintainability issue we are working on more efficiently. For these reasons, it was deemed appropriate to add a qualitative evaluation technique to this study's scope. An Android developer survey and some interviews were conducted within this study's scope as a part of qualitative evaluations. The contents and purposes of these surveys and conferences are discussed in detail in their respective sub-sections below.

3.1.1 Android Developer Survey

The first step of qualitative evaluations in this study is the Android developer questionnaire. Although it is not a method that provides direct benefit in measuring and evaluating maintainability, this method was used to solve the issue of not being up-to-date for the Android field's academic resources, which was criticised in the previous sections. As stated in the literature review subtitle of the previous section, up-to-dateness among existing academic studies stands out as an essential deficiency. For this reason, before evaluating the methods, technologies, and techniques used by Mooncascade's Android team in terms of maintainability, it is considered appropriate to compare the up-to-dateness of these methods, techniques, and technologies concerning current industry trends. To achieve this goal, the Android developer survey mentioned above was used. In addition to evaluating the up-to-dateness of Mooncascade's Android team's methods, this Android developer survey also has the purpose of informing readers about the current Android technology stack. The information collected through this questionnaire can provide an up-to-date resource for researchers who want to work in Android application development. Although this is not one of the main objectives of this study, it is thought that it would be appropriate to have such an addition, given the rapid changes in this study's primary subject and the inadequacy of the current academic literature in this field. Besides, while determining this survey's questions, priority was given to principles and technologies that directly and indirectly affect maintainability, which is the main focus of this study. The questions asked in this survey are listed below. Although the questions are generally prepared to cover the methods used by the Mooncascade Android's team, it can be said that the questions can also include the Android technology stack in a more general context.

- How many years of professional experience do you have as an Android developer?
- Which programming language do you use for Android application development?
- What presentational design pattern do you apply to your Android apps?
- Do you apply Uncle Bob's Clean Architecture to your Android applications?
- Do you follow SOLID principles while developing Android applications?
- Do you follow Uncle Bob's Clean Code principles while developing Android applications?
- What networking library do you use?
- How do you handle asynchronous events in your Android applications?
- What dependency injection library do you use in your Android applications?
- Do you use Android Architecture Components in your Android applications? (LiveData, ViewModel, Room, etc.)

The questions are organised with the help of the Forms application provided by Google Docs. Multiple selections and multiple-choice options have been added so that Android developers can answer them efficiently and in a standard way. The full version of the questions and answers within the Android developer survey's scope, prepared through Google Forms, can be accessed via this web address . Social media has been used to deliver the Android developer survey to as many Android application developers as possible, working in different companies and domains. The Android developer survey has been delivered to Android developers from different companies in different countries through accounts or groups of Android developer communities on social media platforms such as LinkedIn, Discord, Twitter. The author of the study has also shared the survey with many of his colleagues/ex-colleagues working in different companies/countries. Also, the author's ex-colleagues shared the survey with their current colleagues too. To get maximum efficiency from the Android developer survey and obtain valuable and consistent answers, it is envisaged to get 150-200 answers from Android developers. The detailed interpretation of the information and answers collected through the Android developer survey will be made in section 5.

3.1.2 Interviews with Team Members

The second step of the qualitative evaluations carried out within this study's scope is the interviews conducted with Mooncascade's Android team members. Unlike the

Android developer survey that was explained in the previous section, these interviews are designed and conducted to qualitatively evaluate the techniques, technologies, and methods used by Mooncascade's Android team in terms of maintainability. Eight questions were determined to qualitatively measure the effect of the methods used by Mooncascade's Android team on the maintainability of Android applications. Below are listed the questions asked to each member of Mooncascade's Android team during these interviews:

- How many years of experience do you have as an Android developer? Please specify the years in Mooncascade and other companies.
- How many different Android projects have you completed in Mooncascade, and how many different domains did those projects belong to?
- What is your understanding of maintainability in the context of software engineering?
- As an employee of a software development company that provides services to the different domains, what makes maintainability more essential for you?
- What is the importance of maintainability when developing Android applications?
- What is the most critical aspect for maintainability when developing Android applications (e.g. architecture, libraries, programming language, etc.)?
- How do you think the current technology stack of the team impacts Android applications' maintainability? Please specify for each item below:
  - Programming Languages(Kotlin/Java)
  - Software Engineering principles (SOLID/Clean Code)
  - Architecture (MVVM/Clean)
  - Libraries (RxJava, Dagger 2, Apollo/Retrofit)
  - Android Architecture Components (ViewModel, LiveData, Room, etc.)
- What could be improved in our current tech stack and the principles we apply to improve the Android applications' maintainability?

Three main criteria were taken into consideration while preparing these questions. First of all, questions were chosen to measure the participants' background, experience, and understanding of the concept of maintainability in software engineering. Subsequently, questions were drafted about technology and principles that directly or naturally

affect maintainability. Finally, questions were prepared concerning the Android developer questionnaire, which was mentioned in the previous section, to make it possible to compare the results of both the survey and the interviews in the evaluation section. These questions, which are included in the interviews, were directed to the team members in the meetings conducted privately with each team member. The collected answers and information will be interpreted in detail in section 5. It is expected that the knowledge and experience gathered through these interviews will facilitate the accuracy of quantitative assessments and the understanding of validity.

3.2 Quantitative Method

Quantitative evaluation constitutes the most critical part of this study in terms of measuring maintainability. In this study, many other studies on the measurement of maintainability in Android and software engineering were examined. The purpose of these reviews is to find the most appropriate maintainability measurement metrics. Various metrics can be used to evaluate object-oriented software systems in terms of quality and maintainability. The most popular of these metrics are detailed in Barak et al., (2012) [14]. As a result of this examination, it was seen that the concepts of complexity, cohesion and coupling were emphasised in many studies, and it was concluded that the measurements made based on these concepts would be more efficient when measuring maintainability. The effects of these concepts on maintainability have been mentioned in the second chapter, and detailed studies in this field have been referred to. Studies have shown that results retrieved from evaluating these concepts proved to define the level of maintainability [14].

The definition of complexity in software engineering is the difficulty to understand the interactions between the parts of a software system. Higher levels of complexity in software increase the risk of accidentally preventing interactions, increasing the chance of introducing bugs when making changes, thus decreasing maintainability. High coupling between classes also causes complexity when maintaining the software. Changes done in a class reflect the dependent class due to the dependency relationship of the classes. Thus, the software system becomes challenging to maintain. In software engineering, cohesion is how well the methods of a class are related to each other. While the classes' relationship is desired to be loosely coupled, their methods and data fields are desired to be related. Lack of cohesion threatens modularity and software maintenance.

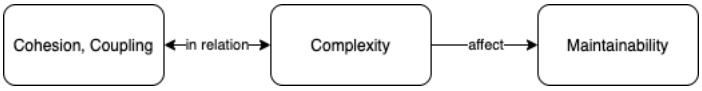

Figure 8. Relationship between cohesion, coupling, complexity and maintainability [14]

As can be seen, complexity, cohesion and coupling are in a tight relationship among themselves, and they all directly affect software maintainability. The figure below shows the relationship between maintainability, complexity, cohesion and coupling. Therefore, based on this result, research was done on measuring the concepts of complexity, cohesion and coupling effectively, and the most suitable metrics for measurement were determined. As a result of this research, five metrics that will enable an object-oriented software system to be evaluated effectively in terms of complexity, cohesion and coupling have been determined. While selecting these metrics, priority has been given to metrics that can handle a software system as a whole in the areas of complexity, cohesion and coupling. This enables the system to be evaluated in terms of maintainability in the most effective way. To emphasise again, although these metrics are used in measuring complexity, cohesion and coupling, they allow maintainability to be measured directly due to the tight relationship between these concepts and software maintainability. These metrics and their intended use are listed below.

- **Weighted Method Count (WMC):** This metric is used to measure object-oriented software systems' complexity. WMC represents a class's cyclomatic complexity, also known as McCabe complexity [34]. It, therefore, portrays the complexity of a class as a whole, and this measure can be used to indicate the maintainability level of the class. The number of methods and complexity can be used to divine maintaining effort. If the number of methods is high, that class is described as domain-specific and is less reusable. Also, such classes tend to be prone to change and defects.

- **Depth of Inheritance Tree (DIT):** This is another metric to measure software complexity. Inheritance increases software reusability; however, one side can create complexity by possibly violating encapsulation since the subclass needs to access the superclass. Furthermore, changes made during maintenance might increase the inheritance tree's depths by adding more children. Therefore, by assessing the inheritance tree available in the product, it is easy to predict how much effort needed to make it stable [14]. It is harder to predict its behaviour if the tree depth is high, and this causes maintenance issues.

- **Number of Children (NOC):** NOC measures the number of descendants of a class, and it is used to measure the coupling level for the corresponding class. NCO also indicates the reusability level of a software system. It is assumed that the number of child classes and the maintainer's responsibility to maintain the children's behaviour are directly proportional. If the NOC level is high, it is harder to maintain and modify the class [35].

- **Coupling Between Object Classes (CBO):** This metric calculates the number of connections to other classes from a particular class, and it is used to measure

coupling. A class is considered coupled if it depends on another class to get its work done [13]. CBO metric is related to the reusability of the class. High coupling makes the code more difficult to maintain because changes in other classes can also affect that class. Therefore these classes are less reusable and less maintainable.

- **Coupling Between Object Classes (CBO):** This metric is used to determine how class methods are related to each other, and it is applied to evaluate cohesion. Cohesion promotes the maintainability of the software systems. High cohesion for a class meant the class is understandable, maintainable and easy to modify [14].

Apart from the above metrics, many other metrics can measure quality and maintainability in object-oriented software systems. The most well-known of these metrics are the Line of Code (LOC) and Halstead Effort metrics. Studies conducted using these metrics in Android, and other software development fields were examined within this study's scope. The methods related to size were not preferred because they were too classical, and they do not give very effective results on maintainability. Halstead complexity metrics were not preferred because they concentrated more on the complexity of classes and methods. Regarding the use of these metrics, Prabowo's work on Android apps' maintainability can be examined [29]. Instead, metrics that facilitate evaluating the quality and maintainability of software systems as a whole were preferred.

After determining the metrics to be used, research has been conducted on how these metrics will be applied. As a result of the research, it has been concluded that metrics can be applied manually or with a tool's help. Since manual implementation will take more time and is prone to error, it has been decided to apply metrics with a reliable tool. Later, a search was done for a static code analysis tool to work with Android Studio and Kotlin programming languages and support the selected metrics. During the research, the CodeMR static code analysis tool drew attention. CodeMR is a powerful software quality tool that is integrated with IDEs and supports multiple programming languages. Java, Scala, Kotlin and C++ [36]. The tool provides an understanding of software quality through its metrics to measure coupling, complexity, cohesion and size. These metrics are often affected by various code characteristics, making them promising for evaluating software maintainability. Besides, the tool provides a visualisation centric approach and generates detailed reports supported by different visualisation options. In this way, it facilitates the application of metrics and makes the results more understandable with detailed reports and advanced visualisation techniques. The tool can also be installed as a plugin in Android Studio and is very easy to use. Apart from that, the tool also makes it possible to measure with many other metrics. The complete list of metrics and other features that the tool offers can be accessed via the tool's documentation. Finally, 2 academic studies conducted using this tool were examined before the tool was started to be used, and information about the tool operation was obtained[37][38]. Finally, the CodeMR static code analysis tool has been chosen to be used in this study due to the

support of Kotlin, its ability to be installed as an add-on to Android Studio, its support for selected metrics, and its advanced visualisation and reporting mechanisms. After this decision, communication was established with the CodeMR team, and a free license was obtained to be used in academic studies.

The use of metrics (CodeMR static code analysis tool) to measure the impact of technologies, methods and principles used by the Mooncascade Android team on maintainability was realised as described following. First of all, a pilot project where metrics can be applied has been determined. While determining the pilot project, the priority was to find a project with an old version developed using outdated technologies, methods, and a weakly structured architecture. However, this project should also have had a new version developed with current technologies and methods mentioned in this study. Thus, using the metrics mentioned above, the impact of the team's current methods on maintainability could be measured. Because of Mooncascade's wide range of realised and ongoing projects, finding a project that met these conditions was possible. Although the full content cannot be disclosed due to the privacy principles, it was possible to apply quantitative evaluation by applying selected metrics to the old and new codebases of an Android application developed very recently and actively in use. In this way, it was aimed to evaluate the effects of the methods, principles and technologies used by the Mooncascade Android team on maintainability at the maximum level effectively. Detailed information about the results and evaluation of the results will be shared in section 5.

3.3 Summary

In this section, detailed information about the qualitative and quantitative evaluation methods performed within this study's scope was shared. The knowledge regarding these methods' contents, why they were preferred, and how they were applied were presented. Thus, the evaluation part's conduction, one of the essential parts of this study, was presented, and the first research question was answered. Detailed results of qualitative and quantitative evaluations will be shared in section 5.

4 Mooncascade Case Study

4.1 Case Company

4.2 Principles

4.2.1 SOLID Principles

4.2.2 Clean code

4.3 Programming Language

4.4 Architecture

4.4.1 MVVM

4.4.2 Clean Architecture

4.5 Libraries

4.5.1 Dependency Injection

4.5.2 Networking

4.5.3 Asynchronous Events

4.5.4 Android Architecture Components

4.6 Summary

5 Evaluation

In this section, the results obtained from applying quantitative and qualitative assessment methods, whose details were shared in section 3, will be presented. As explained in the 3rd section before, the Android developer survey findings and the results obtained from the interviews made with the Mooncascade Android team members, which were applied within the qualitative evaluation scope, will be shared in this section. The quantitative evaluations, which were detailed in the third section, will also be presented in this section. Finally, through the results obtained from the qualitative and quantitative evaluations, the third research question will also be answered in this section.

5.1 Android Developer Survey Results

The Android developer questionnaire results and the findings obtained by interpreting these results are discussed in this section.

5.1.1 Conduction Period of the Survey

The survey accepted answers between April 2020 and March 2021 and reached 164 participants in total.

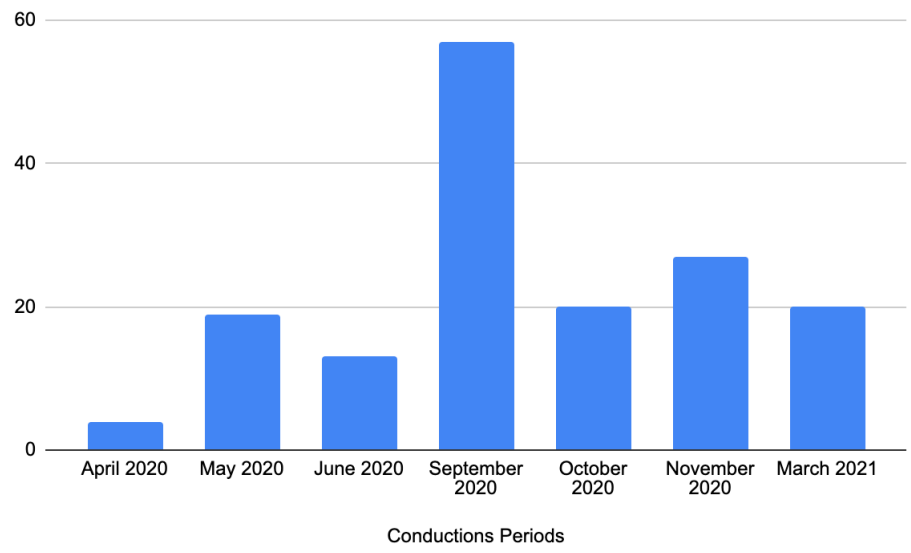

Figure 9. Conduction period results

As shown in Fig. 9, participation in the survey took place only during specific periods of this one-year term. Although the questionnaire accepted answers for a year-long, it was boosted in different periods, as shown in the histogram above. It would be logical to consider this situation while examining the response numbers.

5.1.2 Participant Background

When the 164 participants of the survey are examined, it can be easily said that the survey level is sufficient in terms of participant diversity. In this kind of survey, it is essential to ensure the participants' diversity to get sufficiently accurate and generally reflective results. The Android developer survey found participants in 5 different countries (Germany, Turkey, Portugal, Estonia and, Russia) through the author's current and former colleagues. The survey was answered by Android developers working in 7 different well-known companies in these countries. Also, as previously stated in the 3rd part, the survey reached answers from randomly chosen Android developers through various social media platforms. In this way, the participants' diversity was increased, and getting better results was ensured.

Also, the first question was added to the survey to learn about the competence of participants. As can be guessed, the probability of getting more accurate results from a survey formed to determine the preferred technology and methods for Android application development is directly proportional to the participants' experience. Results can be seen in Fig. 10 below.

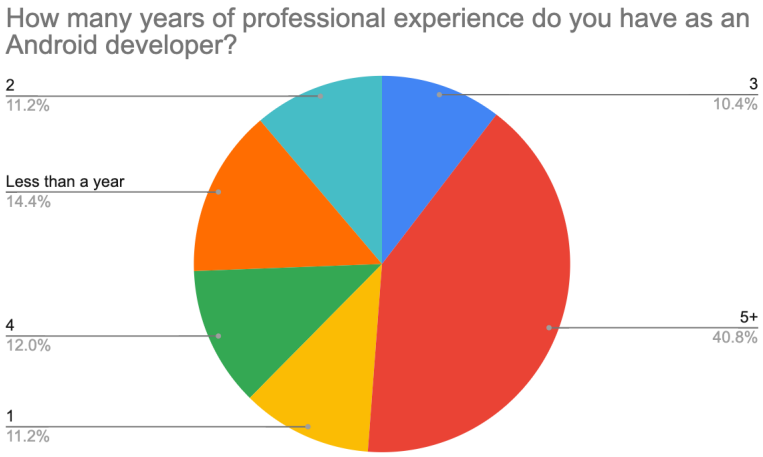

Figure 10. Participant background results

As stated before, the participant diversity was provided to improve the accuracy

of the survey. Also, when the chart above is examined, 63% of the participants have more than three years of experience and these developers can be considered as mid or senior level Android developers and 40% of the participants have five years or more experience. This fact proves that the respondents are Android developers who have sufficient experience and are proficient at Android development. The responses to the rest of the survey questions can be interpreted based on that fact, which is believed to provide more accurate results.

5.1.3 Issues and Limitations

The most severe limitation of the survey was undoubtedly in finding respondents. Although the survey was shared on many different social media platforms and developer community pages, it was able to find very few participants compared to the number of people on these platforms and communities. The initial target for the number of participants was 150-200, and a serious effort was made to reach this number.

Some corrections and filtering were made for the 164 responses collected from these Android developers to avoid off-topic responses and collect non-standard answers under a single topic, thus presenting more consistent results both visually and numerically. For example, for some of the questions, there is the option "other" among the answer options offered, and users can choose this option and enter their answers. In this case, when the data is plotted, results such as the following may occur.

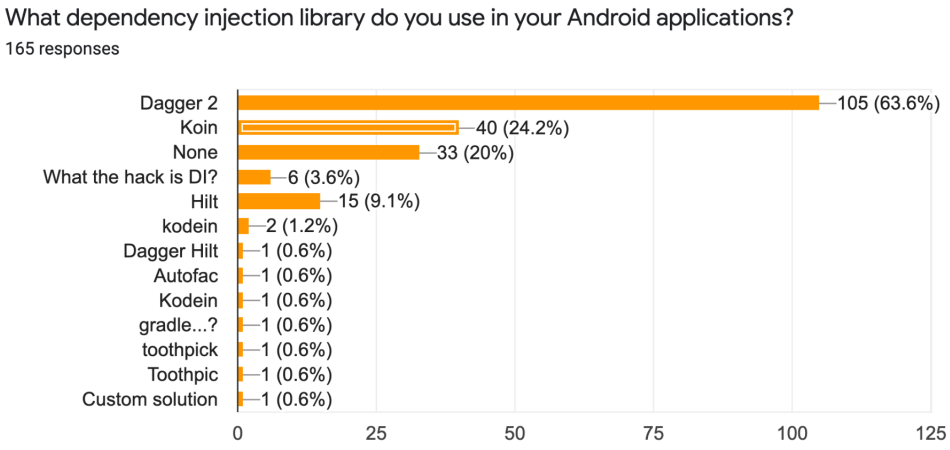

Figure 11. Example chart plotted with non-standardized answers

As shown in the chart, "Toothpick" or “Hilt” libraries do not have a ready-made

response option. Android developers using this library have given their answers in different forms. Thus a chart such as this has emerged. Therefore, it was deemed necessary to arrange the inputs in different forms, which correspond to the same answer.

As an example of the need to filter some answers, a situation in the chart below can be shown. As can be seen in the chart above, which was obtained from the unfiltered survey results, it is seen that developers who develop Android in other forms also participated in the survey. Although it was clearly stated to the participants that the survey included only "Native" Android developers before they filled in the questionnaire, a few such cases were unfortunately not avoided due to the human factor. These kinds of responses have been filtered and edited as they will not contribute to this survey's purpose and reduce the survey's accuracy.

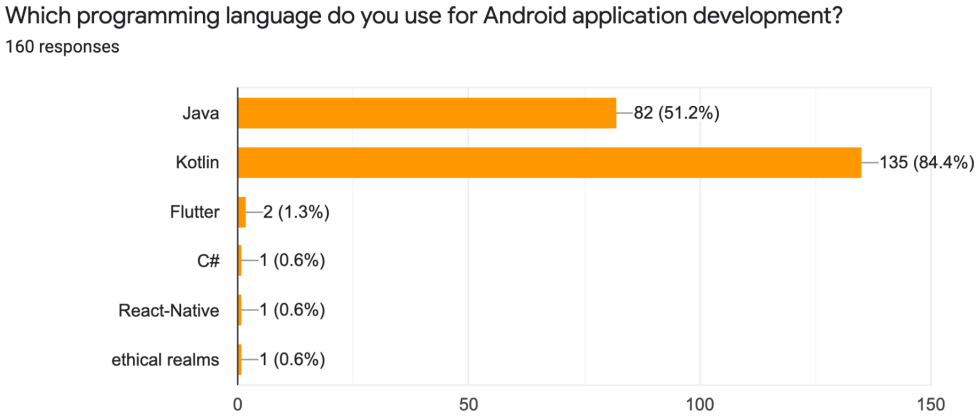

Figure 12. Example chart plotted with corrupted data

The above and other similar situations have been filtered and edited as they will not contribute to the survey's purpose and they reduce the survey's accuracy. Firstly, the survey results were extracted as a Google Sheets file to do this filtering and editing. Then inappropriate data in this file was corrected or filtered, and finally, the charts and numbers were obtained from the accurate data of the file. While applying corrections and filtering, no changes or manipulations were made on the relevant data. Around 20 of the 164 responses were edited to arrange the inputs in different forms, which correspond to the same answer. Besides, five corrupted responses were removed. After the filtering process, charts were created from the remaining 159 responses to obtain more accurate data. The interpretation of the responses is based on this filtered and corrected final version. The charts obtained through filtered and corrected responses and the data obtained from these charts' interpretation are presented below, respectively.

Since it is possible to choose more than one answer for some questions, it should be taken into account that the total number of answers for each question may exceed the total number of participants. Besides, since the first question was added to the survey a bit later than the answers started being accepted, the answers to these questions are less than the rest. While examining the answers, it will be helpful to consider these two situations to prevent confusion.

5.1.4 Programming Language

The second question of the survey asks Android developers the programming language or programming languages they use to develop Android applications. As mentioned in section 2, different programming languages can be used while developing Android applications. Nevertheless, this study focuses solely on "Native" Android development, as mentioned many times before. Therefore, only Java and Kotlin programming languages are among the options offered to answer the second question.

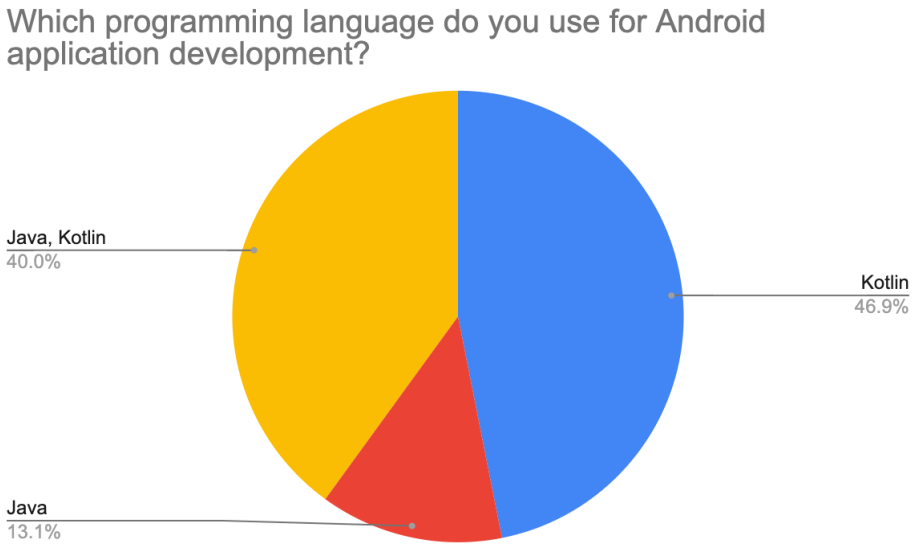

Figure 13. Programming languages results

In Fig. 13 above, Android developers' trends regarding the programming language they use while developing "Native" Android applications can be seen. Around 47% of the Android developers surveyed seem to use the Kotlin programming language. 40% of the participants use Java and Kotlin together, while only 13.1% use the Java programming language. Considering that Kotlin is a programming language suggested by Google and Android and is more "programmer-friendly" than Java, it is not difficult to understand

that the above table is not surprising. As of 2020, Google declared that more than 60% of Android applications were developed with Kotlin. It can be said that the survey results largely overlap with this statement[39]. On the other hand, he fact that some users still use Java can be explained by the existence of Android applications developed with Java before Kotlin was declared as an official programming language for Android.

As stated in section 4.3, Mooncascade's Android team develops Android applications using Kotlin programming language, unless otherwise requested by its customers. When the survey results presented in detail above and the company's choice are compared, it is seen that this choice coincides with the Android community's current trends.

5.1.5 Architecture

The third question of the survey asks participants about their choice of presentational design patterns. In this question, MVVM, MVP, MVI, etc. design patterns were defined as presentational design patterns. The issue of why these design patterns are defined as presentational design patterns rather than architecture and criticisms on this subject can be found in section 2.7. When the results are examined, we come across an exciting and also colourful picture. Below, in Fig 14 the participants' preferences for presentational design patterns can be seen.

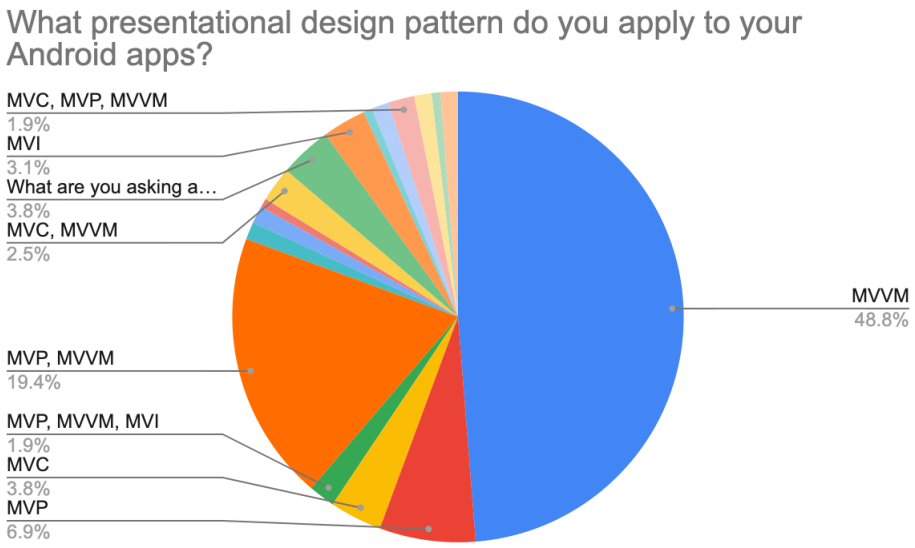

Figure 14. Presentational design patterns results

The first notable conclusion is that almost half of the participants use the MVVM presentational design pattern. Besides, it is seen that another 25% of the participants

stated that they used this design pattern and different design patterns. In other words, a total of 3 quarters of the participants stated that they used the MVVM design pattern in some way or another. Android Architecture Components framework provides some out of box solutions for MVVM. We see that Android developers highly adopt it as of the first quarter of 2021.

On the other hand, the chart presents us that design patterns such as MVP, MVC, MVI are also frequently used. When the participants' responses are sifted through, we see that design patterns such as MVP, MVC and MVI are generally preferred by some developers alongside the MVVM design pattern. These developers have more experience than those who have a single choice of design pattern. In other words, it can be said that as the developer experience increases, the tendency of the developers to choose more than one design pattern also increases. In this case, it can be said that experienced Android developers make the presentational design pattern selection by considering which design pattern will fit the project size and content, rather than what is more popular. As another proof of this situation, it can be shown that developers with 0-3 years of experience have answered this question by selecting the MVVM option. In other words, it is possible to talk about the tendency of Android developers at the beginning of their career to choose popular or "hype" technologies. Another important detail is that 5 out of 6 participants that answered this question as "What are you asking about" had one year or less experience, proving that the knowledge of architecture and design pattern in software development correlates with experience. Lastly, concerning this question, it will be helpful to mention the participants' tendency to choose design patterns such as MVC, MVP and MVI. Comparing the survey results with Figure 7, which is presented in section 2.7 and cited from a study conducted a few years ago in Android architectures, we are faced with similar results despite minor differences. When we look at the comparison results, it is seen that MVVM and MVP were popular among the Android community a few years ago, but MVVM is more preferred today. As mentioned before, it can be said that since the MVVM design pattern started to be provided as an out of box solution by the Google Android team three years ago, this situation increased usage of the MVVM design pattern. In the survey, we also see that 18 of the participants declared that they used the MVC design pattern. Although the MVC design pattern is considered an outdated design pattern in the Android community, the existence of projects developed using this pattern, and considering the suitability of this pattern for small projects; it is understandable why the pattern is still in use. Finally, we see that the MVI design pattern was selected nine times in the last six months (out of 120 answers with a ratio of 0.075). However, it was selected only once (out of 40 answers with a ratio of 0.025) by the participants in the first six months in which the survey accepted answers. This fact is not surprising, given the MVI design pattern's growing popularity during 2020 and 2021. It can be said that this population will increase even more in the upcoming period. As stated in section 4.4.1, Mooncascade's Android team prefers the MVVM

presentational design pattern when developing Android applications. When the survey results (presented in detail above) and the company's choice are compared, it is seen that this choice coincides with the Android community's current trends.

In the following question, users were asked whether they use the "Clean Architecture" structure. Detailed information about Clean Architecture has been given before in section 4.4.2. It is essential to mention why Clean Architecture was asked to the participants differently from the presentational design patterns. Clean Architecture allows the arrangement of an entire application in terms of architecture, unlike the presentational design patterns. The graphical breakdown of participant answers to this question is presented below in Fig 15.

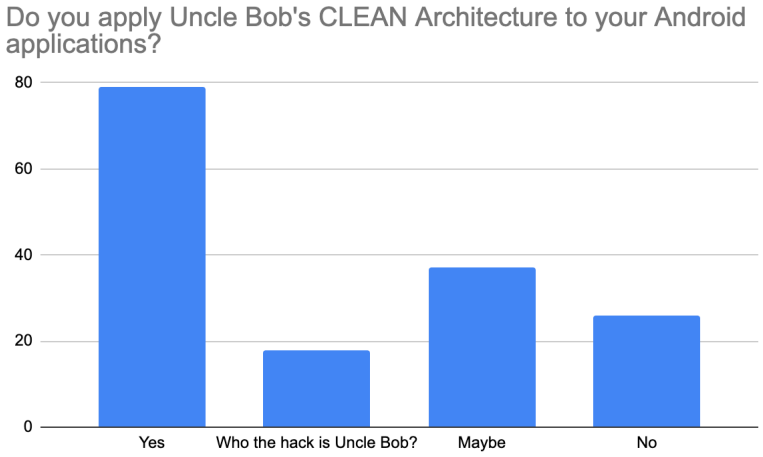

Figure 15. Clean Architecture usage results

When examining the participatory tendencies to use Clean Architecture, it is seen that the majority of the participants adopt this architectural approach. The number of respondents who declared their use of this architectural pattern is almost more than the total of those who declared that they did not or could use it. Besides, 38 of 51 Android developers with five or more years of experience who participated in the survey declared that they use or can use this architectural pattern. Clean Architecture's details, pros and cons were previously shared, but it is widely used among developers, as seen from the survey results. As can be seen in Fig. 7, which is cited from a study on Android architecture carried out a few years ago. Considering the advantages of Clean Architecture, especially when developing large and complex Android applications, and the growing and complexity of Android applications, developers' choice of Clean Architecture makes much sense. Finally, it is possible to say that the Clean Architecture choice of the Mooncascade Android team coincides with the Android developer trends.

5.1.6 Principles

The fifth question of the survey asked the participants whether they follow SOLID principles while developing Android applications. The importance of the SOLID principles and their use requirements were discussed in detail in section 2.4. Results have shown that 66.3% of the participants declared that they follow SOLID principles and 20.6% of them declared that they might follow these principles. Considering how important it is to comply with SOLID principles in software development processes, it can be said that this rate is below expected. 6.3% of the participants stated that they do not apply the SOLID principles, and 6.9% stated that they are not aware of these principles. The figure below contains the graphical breakdown of this data.

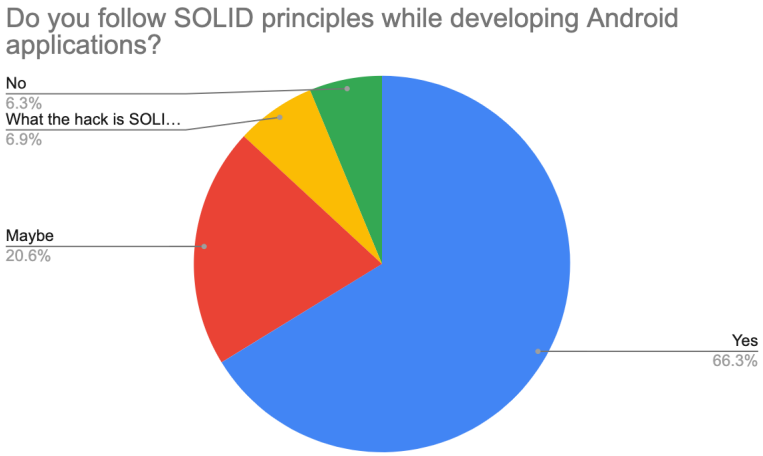

Figure 16. SOLID principles usage results

It is not easy to understand why people who develop software professionally in the Android field or any other field do not want to follow SOLID principles or are not aware of these principles, especially if these people are experienced developers. As stated in chapter 4 before, Mooncascade's Android team actively applies SOLID principles in Android application development processes. This selection is compatible with general Android developer behaviour, as can be seen in the results above.

In the 6th question of the questionnaire, the participants were asked whether they apply the "Clean Code" principles while developing Android applications. The starting point, purpose, advantages and disadvantages of these principles are given in section 4 in detail. According to the results, 75% of the participants stated that they either used or could use these principles. While 13.8% of the participants stated that they do not use these principles, it was observed that 11.3% of the participants were not even aware of these principles. The breakdown of the participants' answers in the form of a pie chart

can be seen below in Fig. 17.

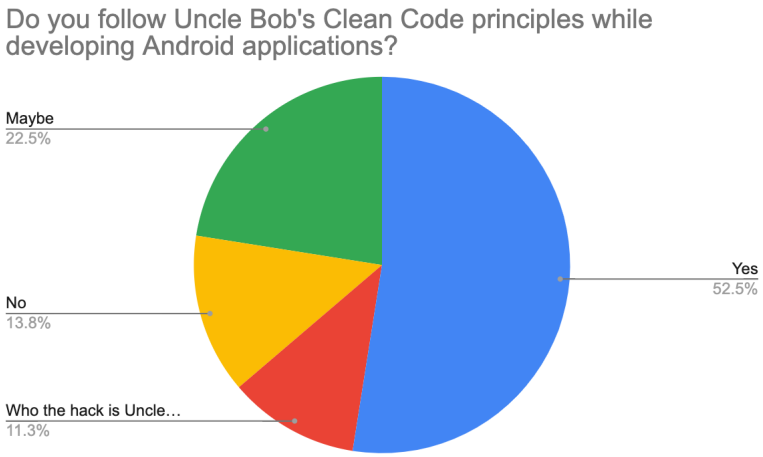

Figure 17. Clean Code principles usage results.

Although there are many advantages of Clean Code principles, discussions are still going on in Android and other software development communities about Uncle Bob and his principles. From this point of view, it can be understood that although most of them actively use these principles, some developers do not. This situation can be interpreted as applying advanced concepts such as Clean Code or SOLID while developing the software directly proportional to the experience. Mooncascade's Android team mainly applies Clean Code principles in Android application development processes. This selection is compatible with general Android developer behaviour when compared to the results above. Further information about how Mooncascade's Android team applies Clean Code principles can be found in section 4.4.2.

5.1.7 Libraries

The next question is designed to ask the participants which networking library they use. Although the network library's use does not directly affect maintainability, this question was included in the questionnaire. It was also among the aims of this study to identify developer tendencies. Also, the use of some advanced networking libraries indirectly affects maintainability due to the out of box solutions they offer. For this reason, it was deemed appropriate to add this question to the survey. When looking at the answers, it is seen that the Retrofit / OkHttp library is dominating. It is observed that 75.6% of the participants use only this library and approximately 13.8% prefer Retrofit/OkHttp libraries and other libraries. The graphical breakdown of responses is presented below

in Fig. 18. It would not be wrong to say that this library is mainly preferred due to its ease when integrating back-end systems running on REST architecture into Android applications. Besides, it is seen that 9.4% (the second-highest rate) of the participants stated that they also used the Apollo library. Apollo, which is the most efficient library used in the integration of GraphQL based back-end systems to Android applications, has the second-highest rate among the answers.

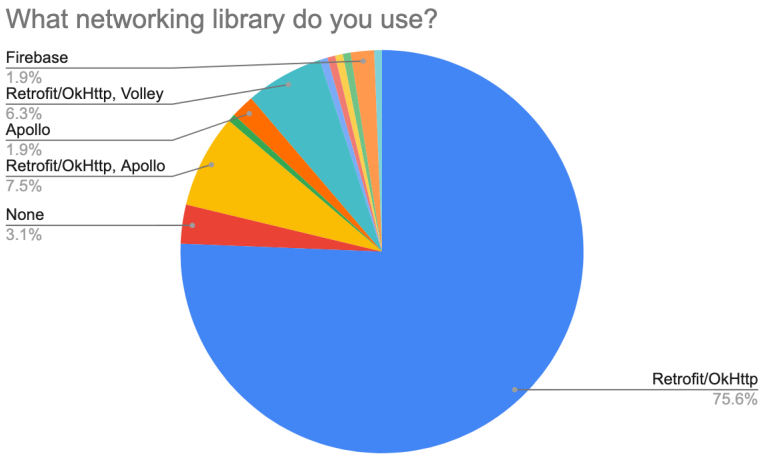

Figure 18. Networking library preferences results

More detailed information and comments about these libraries can be found in 4.5.2. Mooncascade's Android team prefers Retrofit or Apollo libraries depending on the back-end system's type to be used in the project. This preference is in line with the survey results.

The 8th question of the survey asks the question of what libraries they use to manage asynchronous processes while developing Android applications to the participants. This question was included in the survey, considering that many Android applications are based on asynchronous events and the impact of the tools used in managing these events on the application architecture and thus on maintainability. When the results are examined, it is seen that Android developers prefer the Kotlin Coroutines, RxJava and AsyncTask solutions. It is also seen that some of the participants declared that they used more than one solution. The use of more than one solution can be explained by applications that need to be maintained or preferring a solution based on the project needs. Recently, the AsyncTask solution has been deprecated by the Android team. However, it seems that some of the participants continued to use this solution. This situation can be explained by maintaining some previously coded applications using the AsyncTask and are still in use. The use of this solution is no longer recommended [40]. Details of the results can be seen in the below in Fig.19,

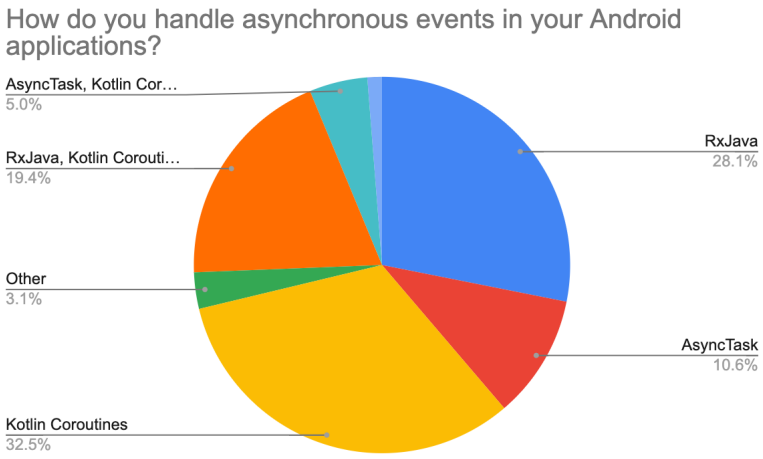

Figure 19. Threading management library preferences results

On the other hand, we see that the Kotlin coroutines solution, which Android officially recommends [41], has the highest percentage in the survey. This solution is increasing among Android developers, as it is easier to learn and use than the RxJava library and because it requires no external dependency. Although RxJava has a steep learning curve and faces the growing popularity of the Kotlin coroutines, it is still preferred by many Android developers for the advanced features it offers. However, there has been a severe increase of applications that have recently migrated their RxJava solutions to Kotlin coroutines[42]. Although the Mooncascade Android prefers RxJava for now, it has been continuing its efforts to switch to Kotlin Coroutines solution. Details on how RxJava is by used are shared in section 4.5.3.

The 9th question of the questionnaire asks the participants which solutions are preferred to apply dependency injection (DI) principles, which significantly impact software maintainability and software architecture when developing Android applications. As shown in Fig. 20, Dagger 2 is the most commonly used DI framework amongst the participant Android developers. Approximately 67.5% of users declared that they used this solution in some way. Besides, Hilt, another DI framework developed based on Dagger 2 by Google's Android team, a relatively new technology, was able to find a place in the survey. Dagger 2 and Hilt are DI frameworks recommended by the Android team. However, it is predicted that Hilt's use will surpass Dagger 2, primarily due to the ease of learning it brings and the decrease in boilerplate code soon [43]. Apart from these solutions, the Koin DI framework also stands out among the results. It can be said that the Koin is preferred among Android developers because of its ease of learning and its ability to get integrated into Android applications with much less boilerplate code when compared to Dagger 2. Also, it is essential to mention that Koin was developed

by using Kotlin programming language. Finally, it is seen that 3.1% of the participants are not aware of the concept of DI and 17.5% of them declared that they do not use any framework for DI. This situation is not surprising given that all of the participants, who were not aware of the concept of DI, had less than a year of experience. Because DI is an advanced software development concept, and its practical implementation is a technique that requires solid experience. It is not mandatory to use any DI framework when developing Android applications. Therefore, it can be mentioned that 17.5% of the participants stated that they do not use any framework and apply their custom solutions. Detailed results can be seen in Fig. 20 below.

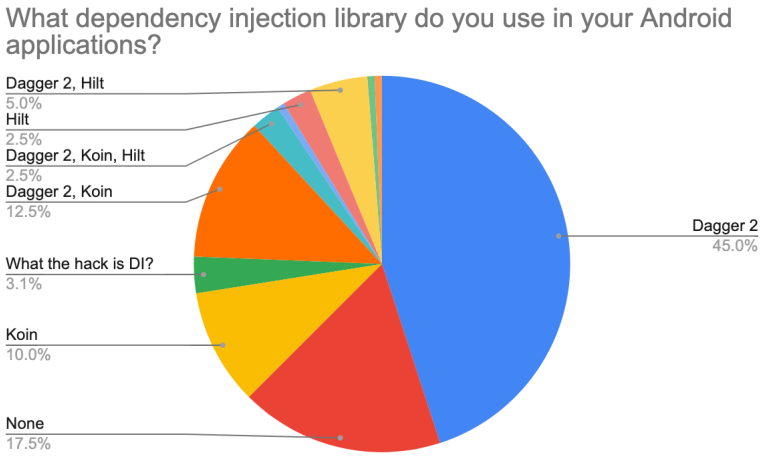

Figure 20. DI Library preferences results

Mooncascade's Android team applies DI principles in their projects and makes these applications through the Dagger 2 framework. Details on how this framework is used are shared in section 4.5.1. The Team is also considering migrating to Hilt soon.

The final question of the survey asks respondents whether they are using the Android Architecture Components framework. When the results are examined, it is seen that more than 92% of the participants stated that they use or can use this framework while only 7.5% of the applicants declared that they do not use it. The high rate of usage is understandable, considering the out of box solutions it offers in solving some of the difficulties encountered while developing Android applications (which were mentioned in the first section, e.g. activity/fragment life-cycle) and the other facilities it provides for Android developers. In addition to this situation, there are groups in the Android community that are distant from this framework because it causes some other difficulties while solving the previously mentioned problems. This claim is controversial, and its details are beyond the focus of this study. However, this may be the reason why some participants do not prefer using this framework. Fig. 21 presents the participant responses

to this question in the form of a column chart.

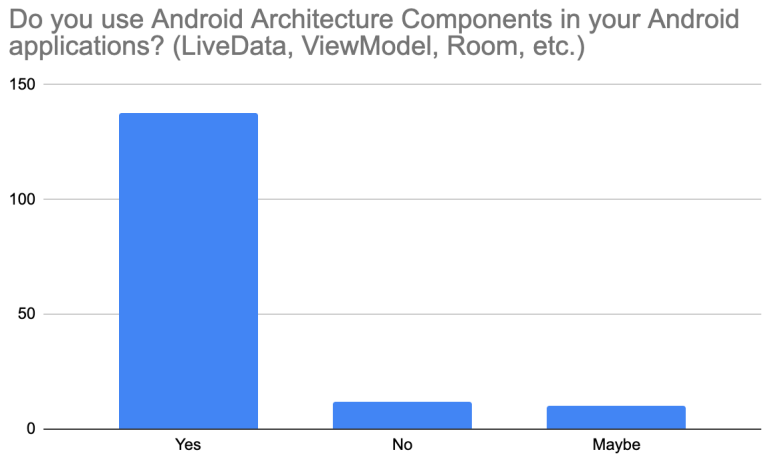

Figure 21. Android Architecture Components usage results

Mooncascade's Android team prefers to use the Android Architecture Components framework. Details on how this framework is used are shared in section 4.5.4.

5.1.8 Summary

The Android developer survey results, which is the first step of qualitative evaluations, are shared in this section. Android developer trends have been identified regarding technologies and principles that are likely to directly or indirectly impact the maintainability identified at the outset. Besides, the up to date developer preferences was observed to avoid an up-to-dateness issue in this study. Not being up-to-date was an issue noticed for similar academic studies, and it had been criticised in the previous sections.

5.2 Interview Results

5.3 Quantitative Evaluation Results

5.4 Summary

6 Discussion

6.1 RQ1 Discussion

6.2 RQ2 Discussion

6.3 RQ3 Discussion

6.4 Limitations

7 Conclusion

References

[1] Taylor Kerns. *There are now more than 2.5 billion active Android devices*. 2019. URL: https://www.androidpolice.com/2019/05/07/there-are-now-more-than-2-5-billion-active-android-device. [Online; Accessed: 25.01.2020].

[2] Anthony I. Wasserman. "Software Engineering Issues for Mobile Application Development". In: *Conference: Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010* (2010).

[3] Ubaid Pisuwala. *How often should you update your mobile app?* URL: https://www.peerbits.com/blog/update-mobile-app.html. [Online; Accessed: 15.02.2020].

[4] URL: http://www.openhandsetalliance.com/oha_members.html. [Online; Accessed: 05.04.2020].

[5] URL: https://source.android.com/. [Online; Accessed: 15.02.2020].

[6] URL: https://developer.android.com/guide/platform. [Online; Accessed: 12.04.2020].

[7] URL: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. [Online; Accessed: 13.04.2020].

[8] URL: https://developer.android.com/guide/components. [Online; Accessed: 14.04.2020].

[9] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professionaly, 1999.

[10] Yegor Bugayenko. *Elegant Objects*. Createspace Independent Publishing Platform, 2017.

[11] IEEE. "IEEE Standard Glossary of Software Engineering Terminology". In: *Institute of Electrical and Electronics Engineers, New York, 1990* (1990).

[12] Jussi Koskinen. "Software Maintenance Costs". In: *Information Technology Research Institute, ELTIS-Project University of Jyväskylä* (2010).

[13] Ahmad A. Saifan and Areej Al-Rabadi. "Evaluating Maintainability of Android Applications". In: *The 8th International Conference on Information Technology ICIT 2017At: Jordan Volume: 8* (2017).

[14] A. Bakar et al. "Review on Maintainability Metrics in Open Source Software". In: *International Review on Computers and Software* (2012).

[15] Hadeel Alsolai and Marc Roper. “Application of Ensemble Techniques in Predicting Object-Oriented Software Maintainability”. In: *EASE ’19: Proceedings of the Evaluation and Assessment on Software Engineering* (2019).

[16] J. R. Mckee. “Maintenance as a Function of Design”. In: *Proceedings AFIPS, National Computer Conference, Las Vegas* (1984).

[17] Capers Jones. “The Economics of Software Maintenance in the Twenty First Century”. In: *Unpublished manuscript* (2006). URL: http://citeseerx.%20ist.%20psu.%20edu/viewdoc/summary.

[18] Robert Cecil Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, Inc., 2009.

[19] Intan Oktafiani and Bayu Hendradjaya. “Software Metrics Proposal for Conformity Checking of Class Diagram to SOLID Design Principles”. In: *2018 5th International Conference on Data and Software Engineering (ICoDSE)* (2018).

[20] W. L. Hursch and Cristina Videira Lopes. “Separation of Concerns”. In: *Technical report by the College of Computer Science, Northeastern University* (1995).

[21] P. Tarr et al. “N degrees of separation: multi-dimensional separation of concerns”. In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)* (1999).

[22] Robert Cecil Martin. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. Pearson Education, Inc., 2018.

[23] Jr Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 1975.

[24] Dewayne E. Perry and Alexander L. Wolf. “Foundations for the Study of Software Architecture”. In: *ACM SIGSOFT Software Engineering Notes* (1992).

[25] Martin Fowler. *Software Architecture Guide*. 2019. URL: https://martinfowler.com/architecture/. [Online; Accessed: 16.04.2020].

[26] Philippe Kruchten. *Philippe Kruchten, The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 2004.

[27] URL: https://www.comp.nus.edu.sg/~damithch/quotes/quote208.htm. [Online; Accessed: 16.04.2020].

[28] David Garlan. “Software Architecture: a Roadmap”. In: *ICSE ’00 2000* (2000).

[29] Ginanjar Prabowo, Hatma Suryotrisongko, and Aris Tjahyanto. “A Tale of Two Development Approach: Empirical Study on The Maintainability and Modularity of Android Mobile Application with Anti-Pattern and odel-View-Presenter Design Pattern”. In: *2018 International Conference on Electrical Engineering and Informatics (ICELTICs), Sept. 19-20, 2018* (2018).

[30] Hugo Kallström. “Increasing Maintainability for Android Applications”. In: *Unpublished master’s thesis, Umea University, Department of Computing Science* (2016).

[31] Roberto Verdecchia, Ivana Malavolta, and Patricia Lago. “Guidelines for Architecting Android Apps: A Mixed-Method Empirical Study”. In: *2019 IEEE International Conference on Software Architecture (ICSA)* (2019).

[32] V. Garousi, M. Felderer, and M. V. Mäntylä. “The Need for Multivocal Literature Reviews in Software Engineering: Complementing Systematic Literature Reviews with Grey Literature”. In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, 26:1–26:6* (2016).

[33] R. Ogawa and B. Malen. “Towards Rigor in Reviews of Multivocal Literatures: Applying the Exploratory Case Study Method”. In: *Review of Educational Research 61(3):265-286* (1991).

[34] T. J. McCabe. “A complexity measure”. In: *IEEE Transactions on Software Engineering (Volume: SE-2, Issue: 4, Dec. 1976)* (1976).

[35] Shyam R. Chidamber and Chris F Kemerer. “Towards a Metrics Suite for Object Oriented Design”. In: *ACM SIGPLAN Notices* (1991).

[36] URL: `https://www.codemr.co.uk`. [Online; Accessed: 21.03.2021].

[37] Jacinto Ramirez Lahti, Antti-Pekka Tuovinen, and Tommi Mikkonen. “Experiences on Managing Technical Debt with Code Smells and AntiPatterns”. In: *4th International Conference on Technical Debt (TechDebt 2021)* (2021).

[38] Jacinto Ramirez Lahti. “Reversing Entropy in a Software Development Project: Technical Debt and AntiPatterns”. In: *Master’s thesis, University of Helsinki, Faculty of Science* (2021).

[39] *Develop Android apps with Kotlin*. URL: `https://developer.android.com/kotlin`. [Online; Accessed: 18.03.2021].

[40] *AsyncTask*. URL: `https://developer.android.com/reference/android/os/AsyncTask`. [Online; Accessed: 18.03.2021].

[41] *Use Kotlin coroutines with Architecture Components*. URL: `https://developer.android.com/topic/libraries/architecture/coroutines`. [Online; Accessed: 18.03.2021].

[42] Vasiliy Zukanov. *The State of Native Android Development*. 2019. URL: `https://www.techyourchance.com/the-state-of-native-android-development-november-2019/`. [Online; Accessed: 18.03.2021].

[43] *Dependency injection with Hilt*. URL: `https://developer.android.com/training/dependency-injection/hilt-android`. [Online; Accessed: 18.03.2021].

Appendix

I. Glossary

What to do here?

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

What to do here?

I, **Mustafa Ogün Öztürk**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Evaluating Maintainability of Android Applications: Mooncascade Case Study, supervised by Jakob Mass.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Mustafa Ogün Öztürk
dd/mm/yyyy

date