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:

Android became one of the most comprehensive mobile platforms in the last decade. This comprehensiveness also brought more challenges to the Android application development. Android's nature, demanding business needs, the frequent update rate of Android applications, and lastly, changing development teams are the four major challenges for Android applications. Maintainability is defined as how easy it is to update, modify, and maintain software. At this point, maintainability emerges as a key concept because developing maintainable Android applications facilitate the above-mentioned difficulties. The primary goal of this study is to evaluate the impact of the technologies and the methods used to develop Android applications by Mooncascede, a software product development company, on maintainability. These methods and technologies include principles (e.g. Clean Code, SOLID), architectural/design patterns (Clean Architecture, MVVM), and third-party libraries (RxJava, Dagger 2 and so on). The evaluation was conducted using the triangulation strategy, which is a mixed-method approach. Qualitative evaluation was conducted via interviews with the case company's Android team (7 participants) and an Android developer survey filled by anonymous developers (over 150 participants). Also, quantitative evaluation was made via object-oriented software metrics. Study results reveal the positive impact of the evaluated methods and technologies on the maintainability of Android applications while pointing to the need for improvements. Results also indicate the need for a new maintainability model specific to the Android applications.

Keywords: Android, Maintainability, Object-Oriented Metrics, Software Engineering

CERCS: P170 Computer science, numerical analysis, systems, control

Androidi rakenduste hooldatavuse hindamine: Mooncascade juhtumianalüüs

Lühikokkuvõte:

Android on saanud viimase kümnendi üheks kõikehõlmavaimaks mobiiliplatvormiks. Samas on see omadus toonud kaasa ka katsumusi Androidi rakenduste arendamisel. Androidi rakenduste puhul on neli peamist proovikivi Androidi olemus, nõudlikud ärivajadused, rakenduste sagedane uuendamine ja muutuvad arendusmeeskonnad. Hooldatavus tähistab seda, kui lihtne on tarkvara uuendada, muuta ja hooldada. Praegu on sellest kujunemas peamine parameeter, sest hooldatavate Androidi rakenduste arendamine leevendab eelmainitud probleeme. Uurimuse põhieesmärk on hinnata Androidi rakenduste arendamisel tarvaarendusettevõtte Mooncascade kasutatud tehnoloogia ja meetodite mõju hooldatavusele. Need meetodid ja tehnoloogia hõlmavad põhimõtteid (nt Clean Code ehk puhas kood, SOLID), arhitektuurilisi ja disainimustreid (Clean Architecture ehk puhas arhitektuur, MVVM) ning kolmandate poolte teeke (RxJava, Dagger 2 jne). Hindamisel kasutati mitut meetodit hõlmavat triangulatsioonistrateegiat. Kvalitatiivse hindamise käigus tehti intervjuud uuritava ettevõtte Androidi meeskonnaga (7 osalejat) ja korraldati anonüümne küsimustik Androidi arendajatele (üle 150 osaleja). Lisaks tehti objektorienteeritud tarkvara näitajate kvantitatiivne hindamine. Uurimuse tulemustest nähtub hinnatud meetodite ja tehnoloogia positiivne mõju Androidi rakenduste hooldatavusele, aga ka täiustusvajadus. Samuti selgus tulemustest vajadus uue hooldatavusmudeli järele, mis on spetsiifiline Androidi rakendustele.

Võtmesõnad: Android, hooldatavus, objektile suunatud mõõdikud, tarkvaratehnika

CERCS: P170 Arvutiteadus, arvanalüüs, süsteemid, kontroll

Acknowledgement

First of all, I am thankful to the authors of referenced studies for the knowledge I obtained from them during my master thesis research. Also, many thanks to Fiona Nevzati and Anna Wilczyńska for their valuable suggestions, my supervisor Jakob Mass for his contributions to the design of this study and Berker Demirer and Dan Pavlovič for their support. Lastly, I extend gratitude to Mooncascade for making this research possible and Mooncascade's Android team participating in the qualitative evaluations of my work.

Contents

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. However, the increasing importance of the mobile era also brought more challenges to mobile application development and Android application development. Designing, developing, testing and understanding Android applications is becoming more complex [2]. These challenges can be examined under four major topics. These are Android's nature and its platform-specific components, sophisticated business needs, the frequent update rate of Android applications, and lastly, changing development teams.

Mooncascade is a product development company based in Estonia. The company provides different software development services to its clients, including Android development. With the increasing demand for Android applications, the company has been facing the previously mentioned challenges when providing Android application development services to different customers from different fields during the development process of Android applications. To overcome these challenges, "maintainability" emerges as one of the most important non-functional requirements. In the context of software engineering, "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 [3]. Developing maintainable Android applications is the way to survive in the competitive market time and cost-effectively [4]. However, achieving this goal is challenging and costly [2].

This study aims to evaluate the impact of the methods and technologies used by Mooncascade when developing Android applications in terms of maintainability. Although evaluating software maintainability is not hard, the ways to be used are vast and choosing a suitable method might be challenging. Different methods and software metrics can be used to estimate the software maintainability value [2, 5, 6, 7, 8, 9]. For that purpose, this study follows the triangulation strategy [10]. Both qualitative analyses (via interviews and surveys) and quantitative analysis (via object-oriented software quality metrics) are conducted. This method is preferred to collect data from different resources (academic resources and Android community) for obtaining more coherent results. The

main motivation of this study is to find proper methods and technologies for developing maintainable applications to eliminate the four major problems in Android application development mentioned above. Thus improving developer/team efficiency and providing time and cost-efficient solutions to the clients. The study only focuses on native Android application development.

1.1 Problem Statement

Mooncascade provides software development services, including Android application development. Demand for mobile applications in the industry has been increasing in the last decade. The company has been having an increasing number of requests for Android applications and facing different challenges during the development and maintenance phases of these Android applications. These challenges are detailed as follows:

Android platform complexity: Android SDK is an over-engineered framework that imposes inherent complexity on the developers [11]. It is different from other software due to the application life cycle, methods for accessing resources between objects, multi-threading concept, and the layout creation process [12]. An Android app consists of activities, fragments, services, broadcast receivers, and content providers provided by the Android SDK and controlled by the Android OS. The necessity of working together in harmony for all these components make Android applications complex software systems.

Business-specific complexity: The definition of complexity in software engineering is the difficulty to understand the interactions between the parts of a software system [13]. Android applications get more and more sophisticated to fulfil 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. When this business-specific complexity comes together with the platform-specific complexity mentioned above, it will be vital to implement software engineering processes to ensure the development of high-quality Android applications [14].

High update rate: Android applications have a high update rate [15] because of bug fixes and the frequent addition of new features based on the changing business requirement and user needs. Thus, Android developers need to keep their applications maintainable. Because it gets harder to maintain Android applications as the codebase grows or changes [2]. For that reason, Android applications should be developed in a way that modifications for new features and bug fixing can be done smoothly.

Changing Development Teams: Developers of an Android application change during the life cycle of the application. Whenever a new developer joins the team, the time required to onboard a new developer to the codebase is directly related to how the Android

applications are developed. Therefore, Android applications must be implemented in an orderly fashion that enables developers to quickly read and understand the app's purpose.

The team aims to develop maintainable Android applications to solve these problems and have internalized some methods and technologies for this purpose, such as coding conventions/principles (e.g. SOLID, Clean Code, Clean Architecture) and using some third-party libraries (RxJava, Dagger 2, etc.). Although these methods and technologies are widely known by the Android community and are believed to be useful for maintainability, their impact on maintainability is empirically unknown. Thus, to find a suitable method for maintainability evaluation and then evaluate the impact of these tools, techniques, and technologies used by Mooncascade's Android team in terms of maintainability, this study will answer the following research questions.

- **RQ1:** What is the proper way to evaluate the maintainability of the Android applications?
- **RQ2:** What is the impact of the methods and technologies used by Mooncascade to develop Android applications on the maintainability of Android applications?

To answer the first research question, research is conducted to find a suitable method for evaluating the maintainability of the Android applications. During this research, previously conducted studies in software maintainability are examined, and efforts are made to find the most suitable method for this case study. The study aimed to answer the second research question using the maintainability evaluation methods obtained from answering the first research question. For this purpose, the obtained methods are applied to two different codebases, one developed by the methods used by the case company and the other not developed by following a certain fashion.

1.2 Contribution

The methods and third-party libraries used in the development of Android applications, which are mentioned in this study, are well-known by the Android community. However, there is not enough empirical data about the impact of these practices and third-party libraries on an important software development concept such as maintainability. The author contributes to understanding how Android application codebases can be measured in terms of maintainability and providing empirical data on the impact of the well known Android development methods and technologies on maintainability.

1.3 Thesis Outline

The rest of this document is organized as follows. In Chapter 2, brief information about the Android platform and Android SDK is given. The chapter continues with key concepts for the maintainability of software systems. Lastly, the methods and technologies used by Mooncascade's Android team to tackle the problems mentioned in the first chapter are presented. In Chapter 3, why and how the literature review was carried out within the scope of this study and the results of this literature review is shared. Also, a brief analysis of the results is presented. In Chapter 4, the methods used to achieve the primary goal of this study are presented. The contents of the survey and interviews are discussed in detail. Also, metrics used while evaluating methods and technologies used by Mooncascade's Android team in terms of maintainability are explained. Thus, the first research question will be answered in this chapter. In Chapter 5, the results for the impact of the practices that are used by Mooncascade's Android team on the maintainability of Android applications are shared. The results obtained from the Android developer survey, interviews with the case company's Android developers and evaluation with the object-oriented metrics are presented as well. Chapter 6 presents the discussion and interpretation of answers to the research questions and evaluation results obtained in the previous chapters. Outcomes of the evaluations will be shared. Also, the limitations of the study will be mentioned. The last chapter concludes this thesis, and future research opportunities are discussed.

2 Background

This chapter covers the key software engineering and Android concepts. Also, the way of Android development at Mooncascade is presented. The chapter aims to facilitate the understanding of maintainability, which is the focal point of this study.

2.1 Android Platform

2.1.1 Android OS

Android is an open-source operating system for mobile devices such as mobile phones, tablets, IoT devices and so on. The Android project 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[1]. 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.

2.1.2 Fundamentals of Android Applications

Android applications consist of components that are essential for the building blocks of an Android app. These components are also an entry point through which the system or a user can enter your app. The components might depend on other compments[2].

Java and Kotlin programming languages can be used to develop native Android applications. There are also other ways of developing Android applications, such as cross-platform solutions (e.g. Xamarin, React Native, Flutter), but this study only focuses on native Android application development. Native Android development means creating Android applications that run on Android-powered devices by using the Android Software Development Kit. Native Android applications consist of XML resources, images, data files, classes and so on. These fundamental components can be listed as follows:

- **Activities:** In the Android environment, the term "activity" refers to the interaction entry point of Android applications for the end-user, a screen with a user interface.

[1] `http://www.openhandsetalliance.com/oha_members.html`
[2] `https://developer.android.com/guide/components/fundamentals`

Activities are represented by the “Activity” class of the Android SDK. Each activity of an Android application is implemented as a subclass of the Activity class.

- **Services:** In the context of Android, "service" refers to 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. Android services can be used to perform background operations such as network operations, content provider interaction, I/O processes, and playing music. In Android, every service must be implemented as a subclass of the Service class that the Android SDK provides.

- **Broadcast receivers:** This term 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. Broadcast receivers are also entry points to the Android applications. 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 SDK provides the "BroadcastReceiver" class and each broadcast receiver must be implemented as a subclass of this provided class.

- **Content providers:** 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. Other apps can query or modify the data if the content provider allows it through the content provider. To the system, a content provider is an entry point into an app for publishing named data items identified by a URI scheme.

Although not a main component, Fragments also has an important place in Android SDK. 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. Fragments are represented by the “Fragment” class of the Android SDK. In order to create a fragment, a class must inherit the Fragment class or existing subclasses of the Fragment class.

Knowing the details of all these components (e.g. implementation details, life-cycle) plays an important role in solving the complications caused by Android components while developing Android applications. Details regarding key Android components and their responsibilities are beyond the scope of this study. For this reason, the summary information in this section has been shared to address these components to raise awareness about their impact on the development of Android applications. Also, examining the official Android documentation in this area can facilitate understanding their nature and their impact on the study topic.

2.2 Key Software Engineering Concepts

2.2.1 Maintainability

"Good programmers write code that humans can understand." - Martin Fowler [16]

A few decades ago programmers were using low-level programming languages that are working close to computer CPUs. These programming languages were designed to be understood by computers, not humans because computers lacked proper hardware, resources, and speed. Priorities back then were different. The computer programs had to be fast and less memory consuming. However, this situation has changed. Today computers are much stronger and software systems are more complex. However this situation also brought some difficulties to software development. Especially when developing large-enterprise software products, not considering how to overcome these difficulties may cause significant failures. In this context, this new reality brought new standards to software development. New programming paradigms were born and the priorities have altered [17].

When the priorities of modern software development are analyzed today, the maintainability of software systems emerges as one of the most critical ones [18]. Maintainability is how well a software system is understandable, repairable, and extendable. In other words, it is a characteristic of software that provides insights into how easily a software system can be maintained [19]. Software systems are born, they live, they change, and eventually, they die. 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. 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. Ignoring all these might cause companies a significant amount of time and money. Developing maintainable software systems is the way to tackle such issues [4]. 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 [20]. When the release cycles of their 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 product from the market and went out of business. Lousy code and, consequently, lack of maintainability was the reason for this company to go out of the business. Considering the changes in software development since the 80s, this example might sound outdated. However, this real-life incident clearly shows how vital maintainability is to software systems and what fatal consequences it can cause if ignored.

The importance of maintainability for software systems can also easily be seen when

looking at its effect on the software development lifecycle and software development costs. A study has shown that the relative expense for maintaining software and dealing with its development speaks to over 90% of its absolute expense [21]. 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 [5]. Reports indicate 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 [22]. The importance of maintainability for software systems is obvious. Also, considering the fast software development lifecycle of mobile applications, the importance of maintainability becomes even more apparent for mobile applications. Facebook's Android application is a good example of this situation[3]. In principle, mobile apps with high maintainability are easier to publish, update and provide high-quality features with less effort. That's why maintenance is considered one of the most important activities for mobile applications [18].

2.2.2 SOLID Principles

SOLID stands for five principles [6]. Following are the brief descriptions of each principle:

- **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

The SOLID Design Principles are object-oriented design guidelines to satisfy software quality attributes such as understandability, modifiability, maintainability and testability. The steps to be taken to increase the maintainability of the software systems can be taken at the design stage. The SOLID Principles are very efficient when it comes to solving such design issues and increasing maintainability of software systems. Not

[3] `https://www.apk4fun.com/history/2430/`

following SOLID principles may lead to serious maintainability problems in the software development lifecycle, such as tight coupling, code duplication, and bug fixing [23].

2.2.3 Separation of Concerns

Software systems have many complex concerns such as persistence, real-time constraints, concurrency, visualization, location control [24]. Software engineering, first of all, aims to increase software quality, lower the expenses of software production, and assist maintenance and development [25]. 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.

Android applications have different concerns such as sustaining limited resource availability on mobile devices, user interface responsiveness, interactions between application components, network connectivity, local data storage, business domain-specific issues, frequent Android platform-level changes and so on. While dealing with these concerns, developers spend a considerable amount of time, and they are diverted from their main goal of building quality Android applications. Eliminating this complexity can be achieved through focusing on the separation of concerns and abstracting away different concerns from each other. Such a goal can be achieved by applying proper software architecture, e.g. "Clean Architecture" [11].

2.2.4 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 [26]

Software systems grow and change over the time. As a software system changes, the interactions between different system elements lead to complexity. To develop reliable software systems that can overcome this complexity, programmers must implement software systems in a generalized fashion. Software systems written in such a fashion can be considered reliably usable, maintainable, testable, and extendable [27]. This fashion is called software architecture. Martin Fowler defines the software architecture as "the shared understanding that the expert developers have of the system design" [28]. The formal definition of 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 behaviour 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" [29]. The role of software architecture in software development can be elaborated under the six major aspects. Those aspects can be listed as follows [30]:

- **Understanding:** Software architecture simplifies the understanding and improves the readability of software systems.
- **Reuse:** Software architecture facilitates the code reuse between the components.
- **Construction:** Software describes the components of the system and dependencies between those components.
- **Evolution:** Software architecture helps developers to understand the consequences of changes more accurately and describes the concerns of the software systems.
- **Analysis:** Software architecture enables analyzing the system consistency, compliance with restrictions imposed by the architecture, and with the quality attributes.
- **Management:** Assessment of software architecture facilitates understanding of risks, requirements, and implementation strategies.

When the list above is examined, the effects of these roles on the maintainability of software systems are obvious. This situation is also clearly demonstrated in a related study. This study has researched Android practitioners, grey literature and white literature, analyzing many relevant resources regarding Android application architecture. Results of the study revealed that the top quality requirement for Android application architecture is maintainability [31].

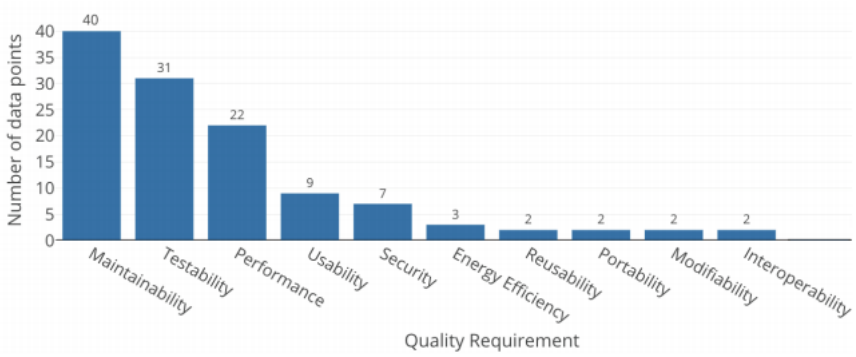

Figure 1. Quality requirement rankings for Android app architecture [31]

Also, other quality requirements presented in Fig. 1, such as modifiability, reusability, and testability, correlate with maintainability [18]. Consequently, the impact of the architecture on the maintainability of software systems is one of the top aspects that should be considered before starting the development.

When architecting Android applications, several different architectural and design patterns are preferred by Android community. Verdecchia et al. (2019) present the developer tendencies in their study [31]. However, considering that their study is from 2019 and the tendencies of Android developers change fast, there might be slight changes in these developer tendencies. Their work also predicted the possible changes.

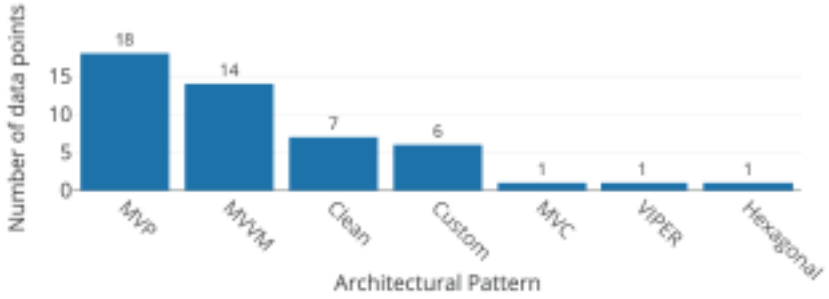

Figure 2. Developer tendencies for Android app architecture [31]

Implementation and theoretical details of such architectural and design patterns are beyond the scope of this study. However, considering the importance of architectural and design pattern choice for the maintainability of Android applications, briefly explaining a few important ones is necessary. As shown in Fig. 2, Model-View-Presenter (MVP), Model-View-View Model (MVVM), and Clean Architecture are the most popular ones. MVP and MVVM are two different derivatives of the MV+X model. They are widely used in GUI-heavy applications. These architectural patterns are built on top of SOLID and separation of concerns principles. They separate application into data, logic and view layers where every layer has its own responsibility [32]. Robert C. Martin introduced Clean Architecture Principles in 2012 [26]. In his book, he indicates that "The goal of software architecture is to minimize the human resources required to build and maintain the required system". Clean Architecture consolidates software engineering practices to produce quality systems with some specifications such as maintainability, testability and so on. The most important characteristic of Clean Architecture is having independent layers. The idea is that nothing in an inward layer ought to rely upon anything in an external layer. Specifically, application and business rules should not rely upon UI, database, moderators, etc. These tenets enable us to create Android applications that are less complex to keep updated as changes in the external circles will not affect the inward ones [33].

2.2.5 Complexity, Cohesion and Coupling

In section 2.2.1 maintainability was defined as how understandable, repairable, and extendable a software system is. To ensure such properties, it is important to know software engineering concepts such as complexity, cohesion and coupling. Because measuring these properties proved to determine the effort of maintainability [7].

IEEE defines software complexity as "the degree to which a system or component has a design or implementation that is difficult to understand and verify [3]. Complexity makes it harder to understand a class; therefore, it gets harder to maintain [34]. Software complexity prevents interactions between different modules, increases the chance of introducing bugs when making changes, thus decreasing maintainability. High coupling and lack of cohesion stand out as the main causes of complexity in software systems. Fig. 3 shows the relationship between maintainability, complexity, cohesion and coupling.

Figure 3. Relationship between cohesion, coupling, complexity and maintainability [7]

Coupling defines the interdependency between the classes. Changes done in one class affects the dependant classes. Software systems with high coupling are hard to maintain. Cohesion is the degree of the relation and interdependency of the members of the class. It promotes software maintainability because high cohesive classes are more understandable, maintainable and easy to modify. Complexity, cohesion and coupling are in a tight relationship among themselves, and they all directly affect software maintainability [7].

2.3 Android Development at Mooncascade

In this section, brief information about the methods and technologies used by Mooncascade's Android team while developing Android applications are presented. Sharing information about these methods and technologies at the baseline level is important for this study because qualitative and quantitative evaluations of their impact on maintainability and obtaining results on these effects are the primary purpose of this study.

First of all, the team uses Kotlin programming language for Android development. It is believed that the use of Kotlin can improve code quality, readability, and productivity

[35]. It is also the supported programming language by Google [36]. For the same reasons the team prefers Kotlin over Java. Also, the team tries to adapt the Clean Code and SOLID principles to their daily programming tasks. Clean Code principles are believed to be to improve the readability and understandability of the code [37]. Also, the application of SOLID principles facilitates the separation of concerns and improves code quality [6]. The team applies these principles for these reasons and tries to maximize their application by code reviews.

When it comes to the architecture of Android application, the team prefers Clean Architecture[4]. Clean Architecture provides a high level of separation of concerns [11] and facilitates changing third-party dependencies and implementation details of the software systems that it is used on. Also, it helps to modify the different layers of the application without affecting the business logic. Thus, it makes Android applications easy to understand, modify and test [38]. The team also uses Model-View-View Model (MVVM) design pattern to present data and applies this design pattern with the help of tools provided by the Android Architecture Components framework published by Google's Android team[5]. With the increase of the scale and the complexity of Android applications, a proper design pattern for the presentation of the data became essential for Android applications to achieve high cohesion and low coupling. Such design patterns enable the different degrees of separation for data, logic and view concerns. Also, an important characteristic for an efficient design pattern is eliminating the bidirectional dependency of views and view models. Thus decoupling of data and view becomes possible, and an important software design objective of high cohesion and low coupling is achieved. The MVVM design pattern provides these requirements for Android applications [32].

The team also uses some third-party Android and Java/Kotlin libraries to develop Android applications. Although such libraries are not a must when developing Android applications, the use of some of them saves Android developers a lot of time and effort. However, the team tends to reduce the use of third-party libraries as much as possible and prefers to use raw solutions whenever feasible. Community support, up-to-dateness, reliability and long-term maintainability criteria are considered when selecting the third-party libraries used. RxJava[6], Dagger 2[7], Retrofit[8], Apollo[9] and Android Architecture Components[5] are the most prominent libraries/frameworks and have the most impact on concepts such as Android application architecture and maintainability. RxJava is a library for designing asynchronous and event-based software applications by using observable sequences. Dagger is a static, compile-time framework used for

[4] https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
[5] https://developer.android.com/topic/libraries/architecture
[6] https://github.com/ReactiveX/RxJava
[7] https://dagger.dev/
[8] https://github.com/square/retrofit
[9] https://www.apollographql.com/docs/android/

dependency injection and it can be used by Java, Kotlin, and Android based applications. Retrofit is used to integrate REST-based back-end systems to Android applications, while Apollo is used for the integration of GraphQL based back-end systems. Lastly, Android Architecture Components are a set of libraries that facilitate designing robust, testable, and maintainable apps.

Lastly, in a related study, it is seen that the results obtained from Android practitioners are similar to the choices of Mooncascade Android team [31].

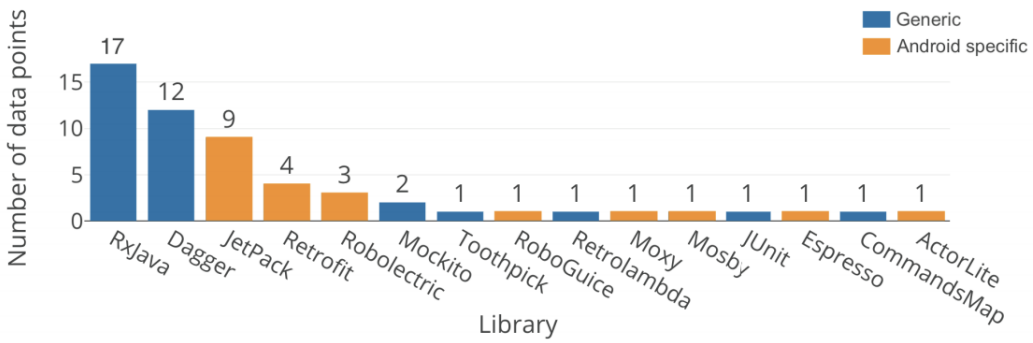

Figure 4. Developer tendencies for Android app architecture [31]

Particularly, the interest of the participants in technologies such as RxJava, Dagger 2, Jetpack and Retrofit can be seen in this study. However, while making a correlation between the mentioned study, which was conducted two years ago, and the preferences of the case company, the rapidly changing nature of the Android ecosystem and changing trends should be taken into account.

3 Related Studies

A literature review was carried out within the scope of this study to examine previous studies conducted on the maintainability of Android applications and obtain more comprehensive information on maintainability measurement. Considering the tight relationship between the topic and industry, Systematic Literature Review (SLR) was not found sufficent 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 [39]. MLR considers resources like blogs, books, articles, academic literature and allows gathering information from academics, developers, practitioners, and independent researchers [40].

3.1 Literature Search Query

To find the related literature a literature search query was formed. The search query was separated into three parts to obtaining more successful findings, individually focusing on a single topic. These parts are Android, maintainability and methods/technologies, respectively. Fig. 5 presents the visualization of the search query.

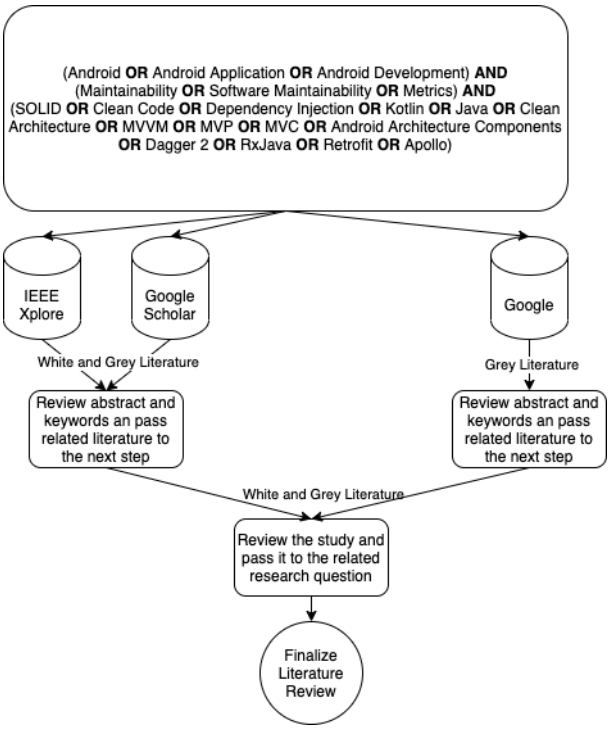

Figure 5. Literature review visualization

A set of criteria has been determined to increase the accuracy of the results obtained from the search query and to filter out possible irrelevant studies among the results. While determining the criteria, issues such as the language of the reviewed studies, their up-to-dateness and their relevance to native Android development were taken into consideration. The inclusion and exclusion criteria are shown in Table 1.

Inclusion Criteria	**Exclusion Criteria**
Studies that discuss the maintainability of the Android applications	Studies that are not focusing on native Android applications
Studies that discuss Android architecture	Studies that are not published in English
Studies that discuss third-party libraries for Android applications	Studies that discuss outdated practices/technologies
Studies that discuss Kotlin usage for Android	Studies with commercial purposes

Table 1. Inclusion and exclusion criteria

3.2 Results

More than fifty studies were found and reviewed as a result of the execution of the search query. Grey literature results are not included in these numbers (e.g. blog posts, official documentation, books). However, the reviewed grey literature was also cited in this study. Moreover, twenty-six studies amongst the literature results were reviewed in detail since they are closely related to this study's field of work. These studies can be grouped according to their topics as follows.

- 9 papers related to Android app maintainability [2, 4, 12, 14, 18, 35, 36, 41, 42].
- 4 papers related to Android app architecture [11, 31, 32, 38].
- 10 papers related to software maintainability [5, 6, 7, 8, 9, 21, 22, 37, 43, 44].
- 3 papers related to software architecture [24, 25, 30].

In addition to the numbers mentioned above, several other studies were reviewed to better understand 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. The literature review also showed that there are many different ways to evaluate software maintainability [7, 8, 9, 43, 44].

Based on these numbers, it can be said that the importance of maintainability for software systems and Android applications is recognized by the researchers. There are

several remarkable studies focusing on the evaluation of the maintainability of Android applications. It would be helpful to refer to these studies previously conducted in this field to understand the topic and be aware of possible limitations. Although the literature review showed that some studies approached the maintainability of Android applications differently, examining the maintainability is similar in most studies.

For example, Ivanov et al. (2020) emphasize the importance of maintainability for Android applications, and they examine the evolution of several maintainability problems during the lifecycle of Android applications. Their results uncover the frequency and evolution of maintainability issues of Android applications and point to the frequent maintainability issues [18].

Hugo Källstrom (2016) conducted a study on the maintainability of Android applications. He implemented three different design patterns (MVC, MVP, MVVM) and evaluated the maintainability of these applications by comparing their performance, modifiability, and testability levels. Results of this study do not provide much insight into the impact of the used design patterns on maintainability [41].

Also, Prabowo et al. (2018) researched the impact of the MVP and anti-pattern approaches on the maintainability of Android applications. They compared maintainability between two applications built with design pattern (MVP) and without design pattern using maintainability metrics such as LoC, Halstead Effort, and Cyclomatic Complexity. Their results showed that usage of MVP increases the maintainability level of the Android applications [42].

Saifan and Al-Rabad (2017) present a different perspective on measuring the maintainability of Android applications by focusing on analyzing a set of Object-Oriented metrics and Android Metrics. Their research reveals that maintainability can be measured in different ways, and the metrics to be used depends on the source code. They also mentioned that Android applications have special features that distinguished them, and to measure the maintainability of Android applications new method is needed [2].

The study of Andrä et al. (2020) draws attention to the most recent research on this subject. They use several different maintainability metrics and match these metrics with different "Clean Code" conventions to evaluate the maintainability of the Android applications developed with Kotlin in class and method level. They tried different static code analysis tools for this purpose. Their results showed that there is currently no suitable tool or plugin for calculating maintainability metrics for Kotlin applications [36].

Panca et al. (2016) evaluate the impact of design pattern selection on the maintainability of Android applications. For that purpose, they implemented different applications using different design patterns such as singleton, memento, state, iterator, factory, builder, and flyweight. They used maintainability metrics such as LoC, Halstead Effort, and Cyclomatic Complexity for maintainability evaluation and compared the maintainability

values of different applications. Their results showed that the maintainability metrics value is increased compared to before using anti-pattern [12].

In addition, the research of Verdecchia et al. (2019) on software architecture choices in Android applications provides insights into the relationship between software architecture, and maintainability [31].

Apart from these studies, some detailed studies on metrics and methods that can be used for the maintainability of software systems are important for understanding the maintainability evaluation. For example, Shyam R. Chidamber and Chris F. Kemerer (1994) proposed a set of software metrics for object-oriented design, respectively, Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of children (NOC), Coupling between objects (CBO) Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM) [8]. The research of Barak et al. (2012) also presents that many object-oriented software metrics can be used to measure the maintainability (including the ones mentioned above) of software. Still, a few metrics are not applicable in predicting the maintainability of this approach [7]. I. Heitlager et al. (2007) identifies the problems of a common maintainability evaluation method, "maintainability Index" (MI) and propose a new maintainability model by defining several requirements for a proper maintainability evaluation method. They also identify foremost maintainability characteristics as analysability, changeability, stability, testability [9].

3.3 Summary

Results of the literature review showed that most studies approach the maintainability of Android applications from a software architecture/design pattern perspective only. Other techniques and technologies can also be used to increase the maintainability of Android applications besides architectural patterns. Still, there are very few studies that focused on these techniques and technologies. The fact that the studies do not focus on these techniques and technologies can be considered a limitation of the academic literature regarding this topic. Apart from this, it is seen that similar metrics and methods are used in many studies in this field. Considering the differences of Android applications compared to other software, it can be mentioned that different methods should be used. In addition, it was not possible to come across studies involving relatively new Android technologies. In some studies, the difficulties of evaluating Android applications developed with a relatively new programming language such as Kotlin in terms of maintainability are also addressed. In general, results indicate the need for a new maintainability model for Android applications.

4 Research Methodology

In this chapter, the methods to evaluate the impact of the techniques and technologies used by the case company on maintainabilty are explained. This study follows the triangulation strategy [10]. This strategy is applied by conducting two different methods within the scope of this study. These methods are quantitative and qualitative evaluations. Quantitative evaluation is made using object-oriented software maintainability metrics, and qualitative evaluation was made via interviews and surveys with Android developers. In this way collecting data from different resources is ensured, and more coherent results can be obtained. Another reason for using this method is that the study subject is closely related to the industry. Due to this close relationship, it is predicted that qualitative evaluations of the Android community and Android developers could increase the accuracy of this study. Thus, maximum efficiency by going beyond the traditional quantitative measurement techniques is aimed. Also, why and how these methods are chosen are explained in this chapter. Hence, the first research question is answered.

4.1 Qualitative Evaluation

When other academic studies dealing with the measurement of maintainability of software systems were reviewed, it was seen that many studies use quantitative methods. When the quantitative evaluations made in these studies are examined, it is seen that Android applications use several object-oriented software metrics while evaluating maintainability. Details of these studies were shared earlier in section 3.2. Although it cannot be claimed that such quantitative measurements are inaccurate, it would not be wrong to say that these measures are inadequate at times. It is essential to make qualitative evaluations and quantitative evaluations to increase the efficiency of the evaluations. In this way, it may be possible to measure developers' experiences and their effects on maintainability. For these reasons, an Android developer survey and interviews were conducted within this study's scope. This section covers these qualitative methods.

4.1.1 Android Developer Survey

The first step of qualitative evaluations in this study is the Android developer survey. The fast evolution of third-party Android libraries and developers trends is vital for Android applications' maintainability because the consistency and stability of principles and third-party libraries affect maintainability. Therefore, this survey was conducted to identify current Android trends and provide support for this study's evaluation. While determining this survey's questions, priority was given to principles and technologies that affect maintainability. Although the questions are generally prepared to cover the

methods used by the Mooncascade Android's team, it can be said that the questions reflect the Android technology stack in general. Also, the first question was added to the survey to learn about the competence of participants. Learning the competence of the participants is important for increasing the probability of getting more accurate results from the survey because the preferred technology and methods for Android application development differs based on the participants' experience. The survey questions are presented below.

- 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. 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, and Twitter. The author of the study has also shared the survey with many of his colleagues/ex-colleagues working in different companies/countries.

4.1.2 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. The interview questions are designed to qualitatively evaluate the techniques and technologies used by Mooncascade's Android team in terms of maintainability. Also, the interviews aim

to determine the importance of maintainability from the case company's point of view. Thus, proving or disproving the study's claim that maintainability is a key concept to overcome the issues mentioned in the problem statement section would be possible. Eight questions were determined for these purposes. 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 get to know about the participants' background and experience. Later, some questions were designed to learn participants' understanding of maintainability in software engineering. Lastly, questions were drafted to learn about participants' thoughts about the impact of technologies and principles used by Mooncascade on maintainability. The interview was conducted privately with each team member. It is aimed that the data gathered through these interviews will increase the accuracy and validity of evaluations.

4.2 Maintainability Evaluation with Object-Oriented Metrics

The literature review results revealed that (see section 3.2) there are different metrics and approaches for measuring the maintainability of software systems. On the other hand, considering the differences between Android applications and traditional software systems, it is obvious that a different maintainability assessment method is required for Android applications. When the problems encountered while developing Android applications, which were mentioned in the first chapter, are examined, it is seen that software characteristics such as software complexity, understandability/readability, modifiability come to the fore. Considering the relationship between maintainability and these characteristics, it was decided that the maintainability measurement to be carried out within the scope of this study should cover these characteristics. To evaluate these software characteristics, it is considered that complexity, cohesion and coupling concepts can be used because software complexity, modifiability, understandability/readability are affected by these concepts. These characteristics and concepts can be matched respectively with each other. As a result of this consideration, it was concluded that the measurements made based on these concepts could be efficient when measuring the maintainability of Android applications. Also, studies have shown that results retrieved from evaluating these concepts proved to define the level of maintainability [7]. Therefore, the maintainability model of this study for quantitative evaluation was formed based on the concepts of complexity, cohesion and coupling and research was done on measuring these concepts effectively. As a result, five metrics were selected for this purpose. 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 to make more efficient evaluations. 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 [43]. 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 [7]. 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. NOC 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 [8].
- **Coupling Between Object Classes (CBO):** This metric calculates the number of connections from one class to other classes to measure the coupling level between classes. A class is considered coupled if it depends on another class to get its work done [2]. 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.
- **Lack of Cohesion of Methods (LCOM):** 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 [7].

CodeMR static code analysis tool has been chosen for this study for the purpose of quantitative evaluation[10]. After this decision, communication was established with the CodeMR team, and a free license was obtained to be used in academic studies. It is a powerful software quality tool that is integrated with IDEs and supports multiple programming languages. The tool provides a set of metrics to measure coupling, complexity, cohesion and size. Besides, the tool provides a visualisation centric approach and generates detailed reports supported by different visualisation options. In this way, it facilitates understanding the results and allows recognition of the bigger picture of the projects from the complexity, coupling, and cohesion point of view. Detailed information on these metrics and assessment methods can be found on the tool's website. The tool can also be installed as a plugin in Android Studio and is very easy to use. Lastly, two studies conducted using this tool were examined to gather more information regarding the tool [45, 46].The CodeMR static code analysis tool has been chosen for this study due to reasons mentioned above. To evaluate the impact of the methods and technologies mentioned in section 2.3 on maintainability, the metrics were applied on two different code bases of the same project via CodeMR. Thus, the impact could be measured by comparing the results from both versions. Although the project's full content cannot be published due to the confidentiality reasons, more information regarding these codebases is shared in the next section.

[10] `https://www.codemr.co.uk`

5 Evaluation

In this section, the results obtained from applying quantitative and qualitative assessment methods, whose details were shared in chapter 4, will be presented. As explained in chapter 4 before, the Android developer survey findings and the results obtained from the interviews made with the members of Mooncascade's Android team, which were applied within the qualitative evaluation scope, will be shared in this section. Results obtained from the evaluation with object-oriented metrics, which were detailed in the third section, will also be presented in this section. Thus, through the results obtained from the qualitative and quantitative evaluations, the second research question will also be answered in this section.

5.1 Android Developer Survey

The Android developer survey results, which is the first step of qualitative evaluations, are shared in this section. Trends have been identified regarding technologies and principles that are likely to impact the maintainability of Android applications.

5.1.1 Conduction Period of the Survey

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

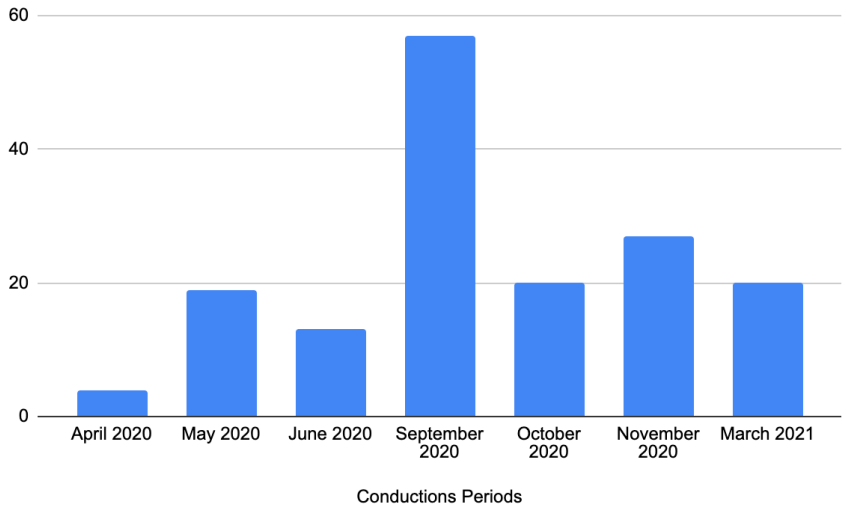

Figure 6. Conduction period results

As shown in Fig. 6, 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 actively advertised in different periods.

5.1.2 Participant Background

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 through the author's current and former colleagues working in seven different well-known companies in different parts of the world. Also, as previously stated in section 4.1.1, the survey reached answers from random Android developers through various social media platforms. In this way, the participants' diversity was increased, and getting more accurate results was ensured. Also, the first question was added to the survey to find out the participant competence. The experience of an Android developer impacts the developer's tendencies when making decisions such as technologies, methods, architectural pattern and so on. Responses gathered from this question can be used to identify such correlations. Results can be seen in Fig. 7 below.

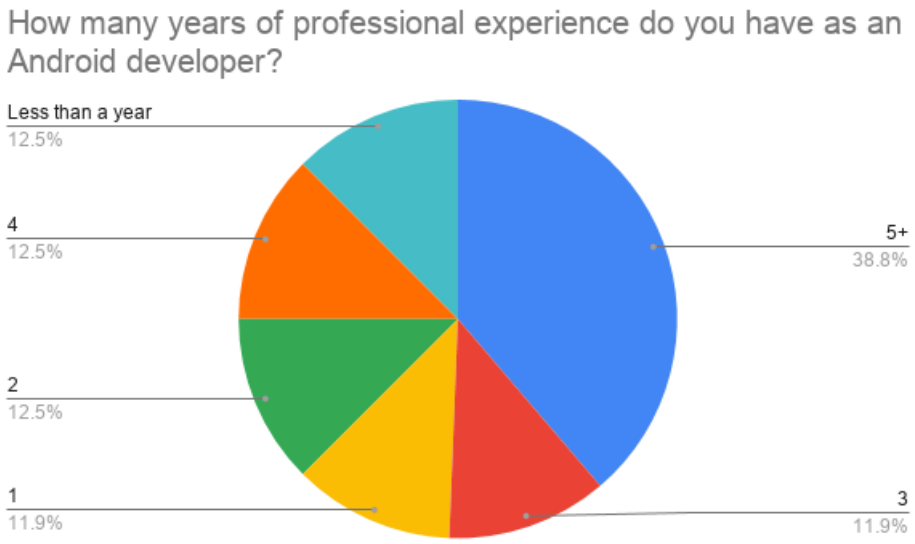

Figure 7. Participant background results

When Fig. 7 is examined, 63% of the participants have more than 3 years of experience while around 39% of them has 5+ years of experience. People with 0-2 years of experience are around 37% of the all participants.

5.1.3 Issues

The most severe limitation of the survey was undoubtedly in finding respondents. Even though 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.

Besides, 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 some of the questions, there is the option "other" among the answer options offered, and users can choose this option and enter their answers. For example, "Toothpick" or “Hilt” libraries do not have a ready-made response option in the survey. Android developers using this library have given their answers in different forms (e.g. Toothpick and toothpick or Hilt, Dagger Hilt or hilt). 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 covers 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 out and edited as they will not contribute to this survey’s purpose and they reduce the survey’s accuracy.

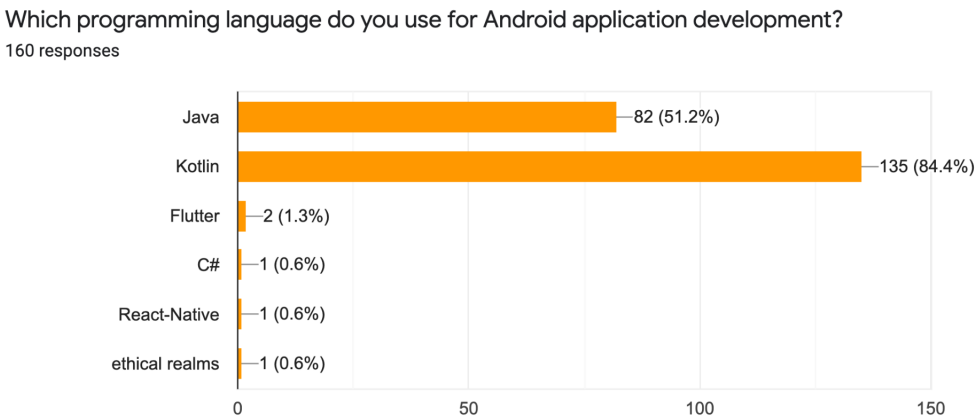

Figure 8. Example chart plotted with corrupted data

The above and other similar situations were filtered out or 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, results were visualised from the filtered and updated 159 responses. 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.

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 2.1.2, different programming languages can be used while developing Android applications. Java and Kotlin programming languages are among the options offered to answer the second question.

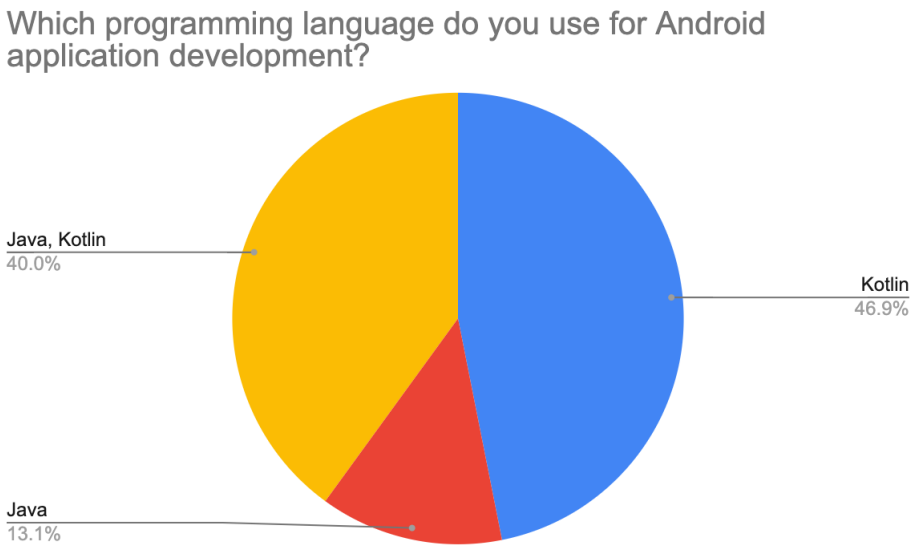

Figure 9. Programming languages results

Fig. 9 presents the Android developers' trends regarding the programming language they use while developing "Native" Android applications. 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.

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. When the results are examined, a diverse set of answers is seen. Below, in Fig. 10 the participants' preferences for presentational design patterns are displayed.

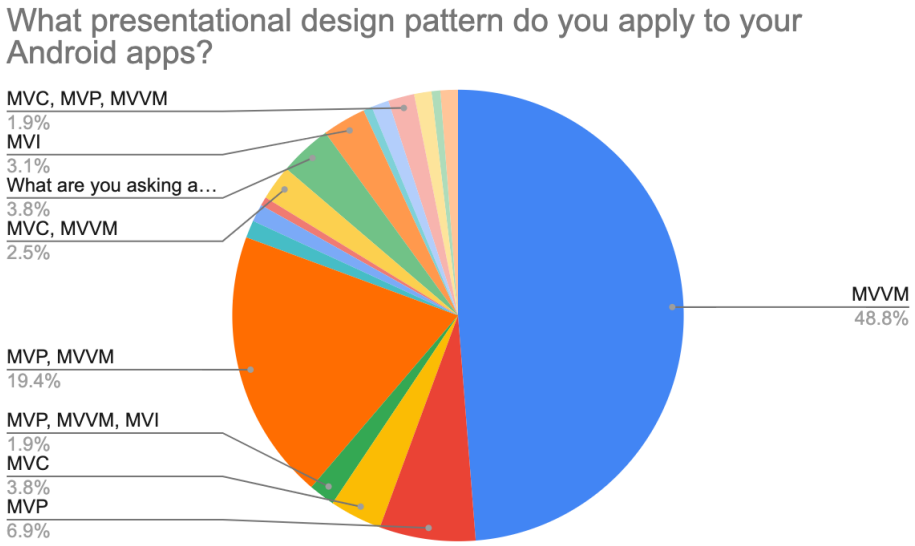

Figure 10. Presentational design patterns results

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. 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, it is seen 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. Also, it was seen that developers with 0-3 years of experience have answered this question by selecting the MVVM option. 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.

In the following question, users were asked whether they use the "Clean Architecture". 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 11. The X-axis represents the options for the question, while Y-axis represent the number of responses.

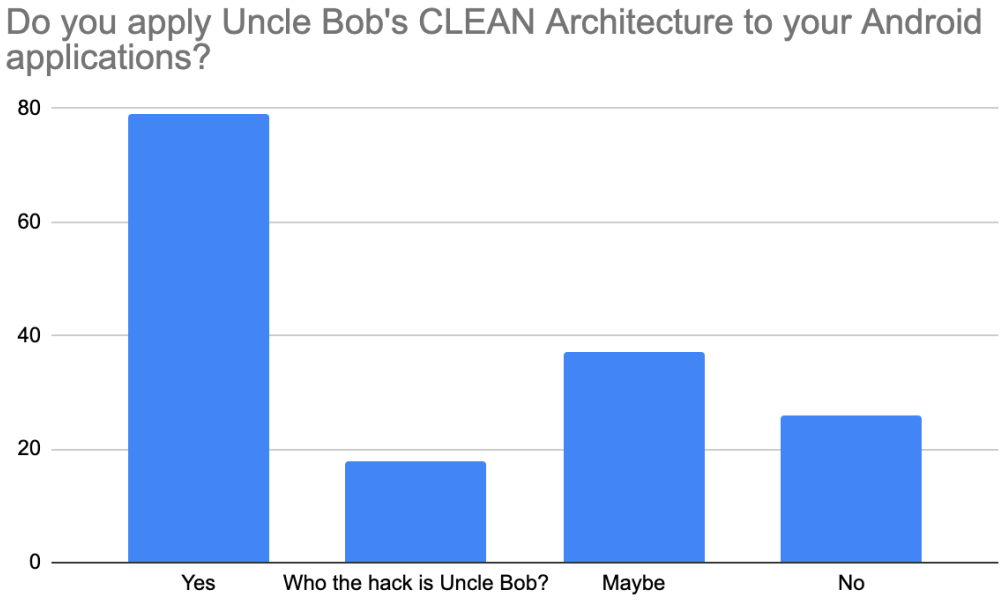

Figure 11. Clean Architecture usage results

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, 50 of 62 Android developers with five or more years of experience who participated in the survey declared that they use or can use this architectural pattern. As a result of obtaining this remarkable data, the subject of whether there is a relationship between the use of Clean Architecture and Android developer experience was examined through the results of the Android

development survey. As a result of detailed examination, a trend like the one below has been caught.

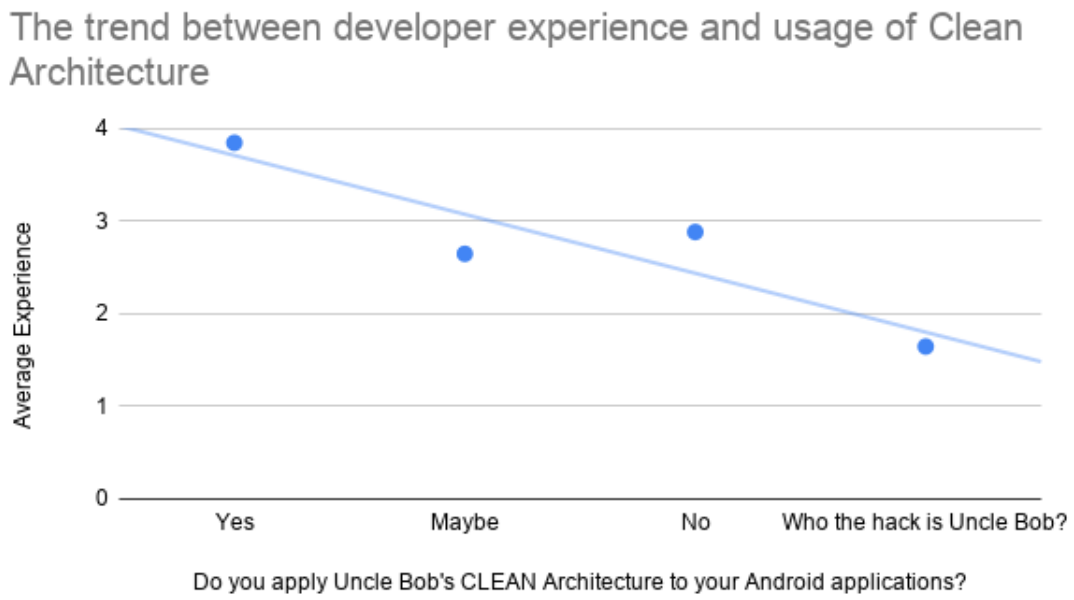

Figure 12. The trend between developer experience and usage of Clean Architecture

As shown in Fig. 12, the Y-axis represents the average experience of Android developers who answered this question, and the maximum value appears to be around 4. When the answer options provided for this question are examined, it is seen that these options are in the form of 0-1, 1, 2, 3, 4 and 5+. Therefore, it would not be wrong to say that the actual average value is more than 4, considering that some participants have 5 years or more of experience. It is important to take this into account when evaluating the results.

As can be seen in Fig. 12, there is a correlation between the usage of Clean Architecture and the experience of the Android developer. While more experienced Android developers seem to tend to use this architecture, there is a tendency not to use this architecture when the developer's experience is low. It is also observed that developers who are at the beginning of their Android development career tend not to be aware of the existence of this architecture.

5.1.6 Principles

The fifth question of the survey asked the participants whether they follow SOLID principles while developing Android applications. 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. 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.

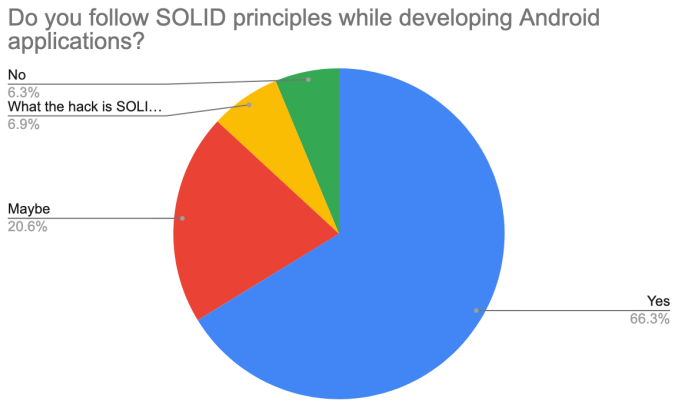

Figure 13. SOLID principles usage results

The 6th question of the survey asks the participants whether they apply the "Clean Code" principles while developing Android applications. The results in the form of a pie chart can be seen below in Fig. 14.

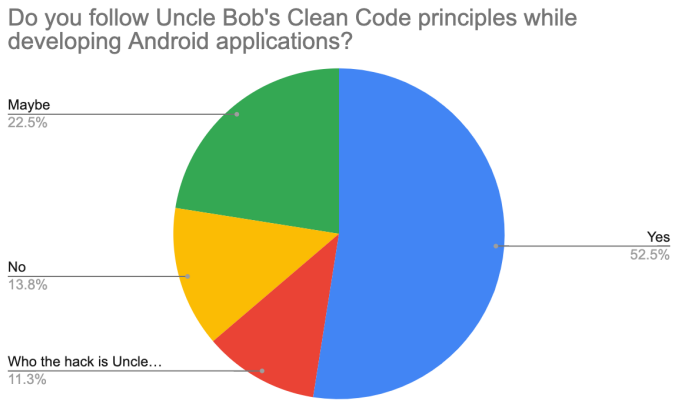

Figure 14. Clean Code principles usage results.

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, and 11.3% of the participants were not even aware of these principles.

5.1.7 Libraries

The next question is designed to ask the participants which networking library they use. The graphical breakdown of responses is presented below in Fig. 15.

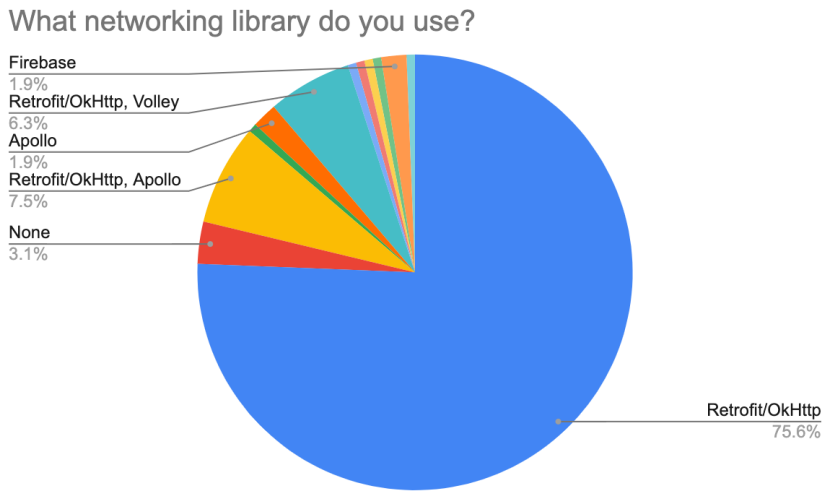

Figure 15. Networking library preferences results

When looking at the results, 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. 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 used library used in the integration of GraphQL based back-end systems to Android applications, has the second-highest rate among the answers.

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. When the results are examined, it is seen that Android developers mostly 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. Recently, the AsyncTask has been deprecated by the Android team. However, it seems that some of the developers continued to use this solution. Besides, it is seen that the Kotlin coroutines, which Android officially

recommends [11], has the highest percentage in the survey (32.5%), which is followed by RxJava (28.1%). Details of the results can be seen in the below in Fig.16.

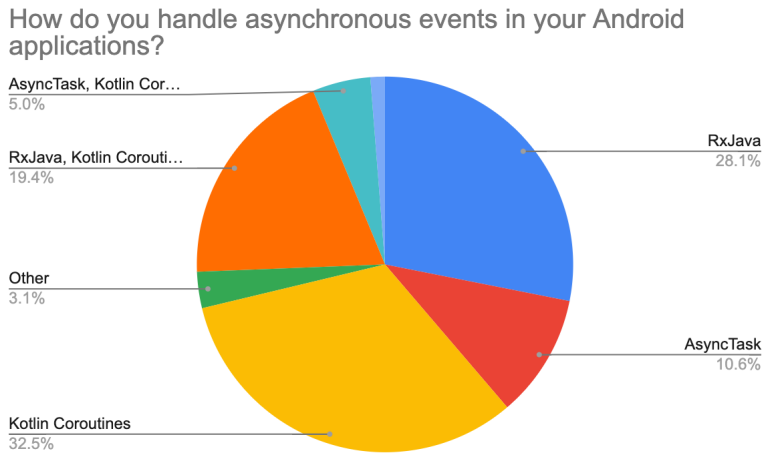

Figure 16. Threading management library preferences results

The 9th question of the survey asks the participants which solutions are preferred to apply dependency injection (DI) principles. As shown in Fig. 17, Dagger 2 is the most commonly used DI framework amongst the participant Android developers.

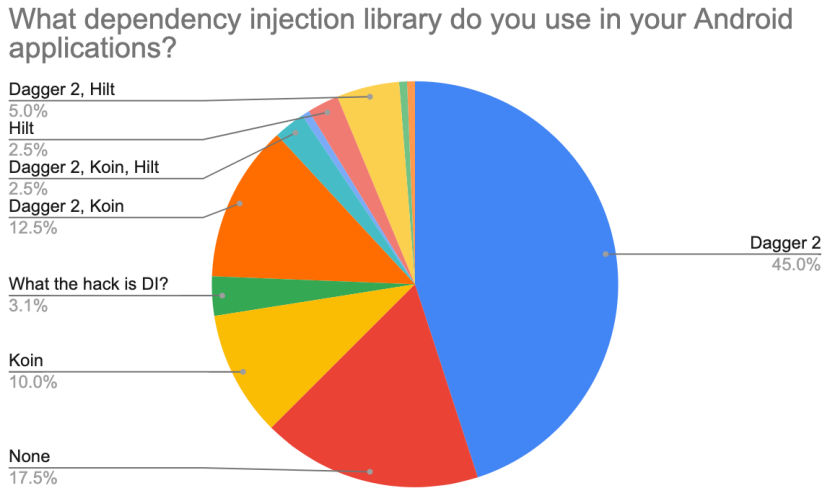

Figure 17. DI Library preferences results

[11] https://developer.android.com/topic/libraries/architecture/coroutines

Approximately 67.5% of users declared that they used this solution in some way. Besides, Hilt, another DI framework developed based on Dagger 2, a relatively new technology, was able to find a place in the survey. Apart from these solutions, the Koin DI framework also stands out among the results. Lastly, 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. Detailed results can be seen in Fig. 17.

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. Fig. 18 presents the participant responses to this question in the form of a column chart.

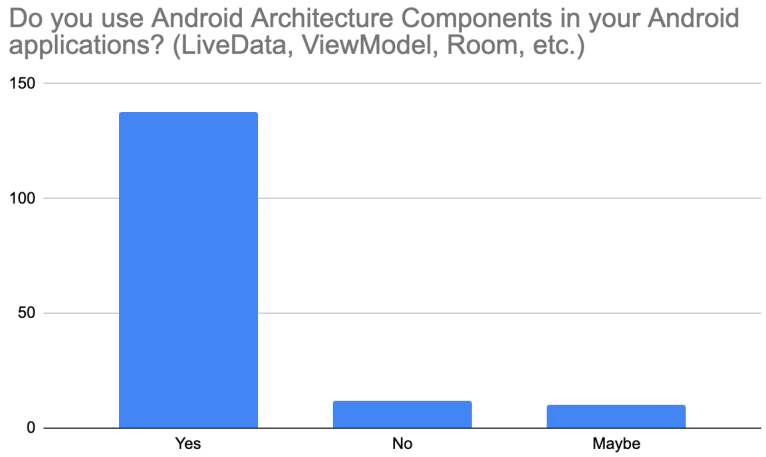

Figure 18. Android Architecture Components usage results

5.2 Interviews with Team Members

In this section, the results obtained from the interviews made with the members of the Mooncascde Android team as a part of the qualitative measurements are shared.

5.2.1 Conduction

Interviews, the details of which are shared in section 4.1.2, were conducted with Mooncascade's Android team members within the scope of qualitative evaluations. Interviews were conducted online, privately with each member of the team. The team was eight in total, and every team member, except the author of this study, participated in these

interviews. The interviews took place in April 2021, and all the responses of the team members were kept in written and video record format. Subsequently, the presentation of the results was realized as a result of the rewording of these records.

5.2.2 Participant Background

The first three questions of the interview were designed to understand the competencies of the participants in the field of Android application development and to see how well their knowledge is regarding the concept of maintainability in software engineering. The average experience that participants have is 5.6 years, ranging from 3 to 8. In addition, the participants stated that they accomplished many Android projects in different fields, and the Android applications implemented through these projects are the applications that are actively used today. Considering the knowledge levels of the participants about maintainability in software systems, it was seen that the participants emphasized the areas of readability, understandability, modifiability and up-to-dateness. Without exception, all the participants highlighted these concepts in their answers to the questions related to the understanding of software maintainability.

5.2.3 Importance of the Maintainability

The fourth, fifth and sixth questions of the interviews are designed to determine the importance of maintainability for the case company.

The fourth question of the interview asked the participants the importance of maintainability for the case company. Five of the participants stated that maintainability is an essential requirement for the company due to how the company works. Since the company works for different customers from different fields and the developer circulation in these projects is high, these participants stated that the maintainability of the projects is critical in terms of cost, time and effort. These participants emphasized that it is vital for the company to develop Android applications with high understandability, high modifiability, and high maintainability due to facilitating the development, hand over of the projects to the customer and maintenance operations required in the long term. Also, two participants stated that maintainability is already critical for software systems and that there is no extra situation that makes it more important for the company.

The fifth question of the interview asked the participants the importance of maintainability for the Android applications. Four of the participants stated that maintainability is important in solving the complexity issues in Android applications. Also, the participants stated that the fact that the Android platform and third-party Android libraries are evolving rapidly makes the maintainability of the Android application crucial. Participants stated that if unstable libraries are preferred for the sake of the industry trends the fast

evolution of the platform may cause problems in terms of maintainability. Also, two participants stated that maintainability is already important for software systems and that there is no extra situation that makes it more important for the Android applications.

The sixth question of the interview asked the participants for the most important matter for Android applications when it comes to maintainability. In general, it was seen that the participants provided two different answers to this question rather than stating a single matter. When the responses of the participants are examined, it is seen that the application architecture is emphasized four times, the third-party libraries emphasized four times, and the documentation once. The reason why third-party libraries affect maintainability was explained in the previous paragraph. It was seen that the developers put the same emphasis when answering this question as well. In addition, it was emphasized by the participants that the choice of architecture increases the maintainability of the applications as it makes the codebases of the applications better structured, standardized and consistent. Lastly, one participant stated that the documenting of the Android codebases improves understandability.

5.2.4 Developer Reviews on Case Company's Methods

The last two interview questions were directed to the participants to qualitatively evaluate the methods and technologies used by the company while developing Android applications (see section 2.3) from the maintainability point of view.

When the responses of the participants are examined, positive comments about Kotlin programming language draw attention. Kotlin was found to be more concise and more readable than Java by the participants, and the participants stated that this situation has a positive effect on maintainability. Also, one participant stated that programming language did not have any positive or negative effect on maintainability.

When looking at the answers about the SOLID and Clean Code principles, again, mostly positive reviews are seen. Participants think that the impact of these principles are positive on maintainability in terms of regulating the coding conventions of their codebases, increasing readability, facilitating separation of concerns and increasing consistency.

On the subjects of architecture and design patterns, the participants stated that they found the MVVM design pattern positive for maintainability for it decreasing the coupling level between the classes responsible for view and presentation. Besides, the support of the Android Architecture Components framework offered by Google's Android team for MVVM and stability of this framework were considered positively by the participants in terms of maintainability. In the case of Clean Architecture, most participants stated that this architecture successfully managed to separate concerns, thus

reducing the complexity and coupling levels of Android codebases, therefore increase the maintainability of the Android applications. However, these participants also stated that this architecture increases the complexity of small and medium-sized projects and decreases the readability of the codebase thus negatively impact the maintainability.

It was observed that the participants made different comments about the effect of the third-party libraries used by the team on the maintainability of the applications. For example, the RxJava library was criticized in different ways. Participants stated that the steep learning curve of the RxJava library decreases the understandability of Android applications. Thus maintainability was negatively affected. In addition, the risk of RxJava becoming obsolete in the near future was also stated by some participants. It was mentioned that this situation had a negative effect on maintainability. On the other hand, some participants stated that this library has positive effects on maintainability due to its strong community, documentation and advanced features. Participants stated that Dagger 2 library has a positive impact on maintainability since it is a reliable and well-documented library supported by the Google Android team as a standard way of dependency injection. On the other hand, it was emphasized by the two participants that the complex structure of this library and its steep learning curve may cause problems in readability and understandability and that maintainability may be negatively affected. Regarding the Retrofit and Apollo libraries, all of the participants stated that these libraries have positive effects on maintainability. Reasons for that mentioned by the participants as strong community support and documentation, stability and reliability. In addition, the participants stated that the Retrofit library had become the industry standard, and it has a positive effect in terms of maintainability as it is easier to use than other alternatives. Architecture Components framework offered by Google's Android team has also found handy by the participants when it comes to the maintainability of Android applications. And as stated before, the reason for that is that the framework was seen as a stable framework by the participants and considered positively by the participants in terms of maintainability. However, one participant has stated that due to some internal issues that this framework has, the framework might badly affect maintainability.

Ultimately, participants also stated that improvements can be made in the currently used methods and technologies. Participants mentioned that the documentation of the Android applications could be improved. They also mentioned that libraries can be reconsidered based on the current trends and up to date technologies (e.g. technologies mentioned in section 5.1 such as Kotlin Coroutines, Dagger-Hilt).

5.3 Evaluation with Object-Oriented Metrics

In this section, information regarding the application of object-oriented metrics and the results obtained from this application are presented.

5.3.1 Sample Codebases

In this section, detailed information about the codebases which the metrics mention in section 4.2 are shared. As mentioned in section 4.2, two codebases belonging to the same project have been selected to apply the metrics. To facilitate the explanation, these codebases will be mentioned in the form of ***CB-1*** and ***CB-2*** in the remainder of the study.

When CB-1 is examined in detail, the following situation is encountered. First of all, it is the Android application's actively used version. The project was started to be developed using the Java programming language. Later, some features were developed using Kotlin programming language. There is no consistent choice of software architecture throughout the application. Although the application consists of 5 different modules, the modules' boundaries are not determined according to a certain standard. Some modules are feature-based, while some are layer-based. A similar situation is observed in packaging. The packaging organization of the application is inferior. While some features of the application have been developed with the MVP design pattern, some features have been developed with MVVM. It is controversial to what extent these design patterns are applied correctly. Static classes and singleton solutions have been used dangerously in practice. Besides, the SOLID principles have been ignored and no dependency injection is applied. It is also worth noting the use of some outdated libraries. Also, when looking at the Git history of the application, the commits of 11 different developers are seen. This situation is critical in describing the developer circulation in the application and explaining its serious organisation problems. The relationship between the developer circulation and the maintainability of software systems was previously mentioned in section 1.1.

CB-2 was developed as part of this study to evaluate the impact of the methods and technologies used by the case company (see 2.3) by comparing it to CB-1, a codebase developed without these methods and technologies. CB-2 is developed using Kotlin programming language. Clean architecture and clean code, and SOLID principles are followed during the development. The application modules are separated based on the layers of different responsibilities (view, presentation, domain, data, local storage, networking), and the packaging is feature-based. While developing the application, maintainable and reliable Android libraries have been used. The organisation level of this codebase is very high and consistent, and it has been arranged to be a standard throughout the whole application. On the other hand, development for this version of the application's is still ongoing, and there are only 4 main features that have already been developed. The features currently developed for CB-2 are splash, login, register and main screen features. In this respect, CB-2 falls short in terms of developed features compared to CB-1. Therefore, the evaluation was only be made over the features that have been developed in both CB-1 and CB-2. The other features of CB-1 were removed from the codebase before the evaluation. Removal of such features was relatively easy

since the developed features are quite independent of the rest of the application.

5.3.2 CodeMR

The CodeMR tool uses different visualization methods in the reports generated by the application of metrics. The charts are created with the help of these different visualization techniques based on the metric values obtained from the analysis conducted by the tool. In this study, the visualization methods known as "Metric Distribution" and "Package Structure" were preferred to present the results of the evaluations. The main reason for choosing this methods is that it makes the visual understanding of the results easier than other methods and also these methods provide a better situational perceptibility for projects. These visualization techniques were used to visualize the data obtained as a result of applying the metrics specified in section 3.2 and the analysis presented by CodeMR on complexity, coupling and cohesion concepts. Other visualization methods and charts are very detailed, and sharing such detailed results is beyond this study's scope. The tool also presents the values for each metric over the provided codebase and presents these values. Values for each metric are presented at project, module, package and class scopes. However, in the evaluations within the scope of this study, only the results of the metrics previously determined and explained in section 4.2 were used and these metrics are Weighted Method Count (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object Classes (CBO), and Lack of Cohesion of Methods (LCOM). Fig. 19 presents the metric values calculated with the help of the CodeMR Intellij IDEA plugin.

Name	Complexity	Coupling	Size	Lack of Co...	CBO	RFC
android						
com.mooncascade.[redacted]	low	low-medium	medium-high	low		
com.mooncascade.[redacted].api	low	low	low-medium	low		
com.mooncascade.[redacted].common	low	low-medium	low-medium	low		
com.mooncascade.[redacted].connectionAdapters	low	low	medium-high	low		
Call	low	low	low	low	4	8
CallCaching	low	low	low-medium	low	3	6
Command	low	low	low	low	1	1
DataUpdateConnectionAdapter	low	low	low	low	3	2
com.mooncascade.[redacted].data	low	low	low	low		
ISQLiteDatabase	low	low	low	low	4	12
SQLDatabase	low-medium	low	low	low	4	16
SQLStrings	low	low	low	low	0	0
WrappedDatabase	low	low	low	low	4	23

Figure 19. CodeMR Metric Value Presentation

There are two important points to be aware of when interpreting the charts created by these methods using CodeMR. The first of these points is legends used to indicate metric levels. As can be seen in Fig. 20, each colour corresponds to a metric level that is

represented by a certain metric value threshold. For the metrics selected for this study, low values tend to indicate better results while high values point to the contrary.

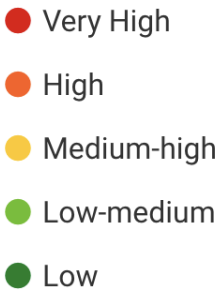

Figure 20. CodeMR Metric Level Indicators

The second important point is how the percentages on these charts are calculated while realizing the charts created by CodeMR. There might be different metric levels in different percentages in the charts. These percentages of metric levels for the selected metrics are illustrated in charts proportionally to the code size of classes in this level. Detailed information about the formulas used in calculating metric values and how the values are interpreted amd visualized can be accessed through the CodeMR documentation[12].

Presentation of the results obtained via CodeMR is done as explained below. The charts and the metrics values obtained from the evaluation are shared. Although these techniques can be applied at the class and package level, it was preferred to present the project level results. It would not be very effective and useful to present these results for each class and package since there are many classes and packages for both codebases. On the other hand, at the class level, for making some evaluation and comparison, a few sample classes with high functionality were selected from both codebases, and they were evaluated and compared. Also, in accordance with the confidentiality requirements, the package and class names that will call the application name were hidden in the shared analysis results and figures. Then the results were presented via visualisation methods and numeric values. Following sections present the evaluation results and findings.

5.3.3 CB-1 Results

When the numerical analysis results performed on the CB-1 via the CodeMR tool are examined, it is seen that the tool analyzes 2079 lines of code belonging to this codebase. These lines of code belong to 118 classes in 12 different packages of 4 different features which were mentioned in section 5.3.1. In Fig. 21, a CodeMR table that reflects the

[12] `https://www.codemr.co.uk/documents/`

general situation of CB-1 is shared. In this table, a grouped overview of the complexity, coupling and cohesion levels determined through the metric values obtained from the analysis of the codebase is shown.

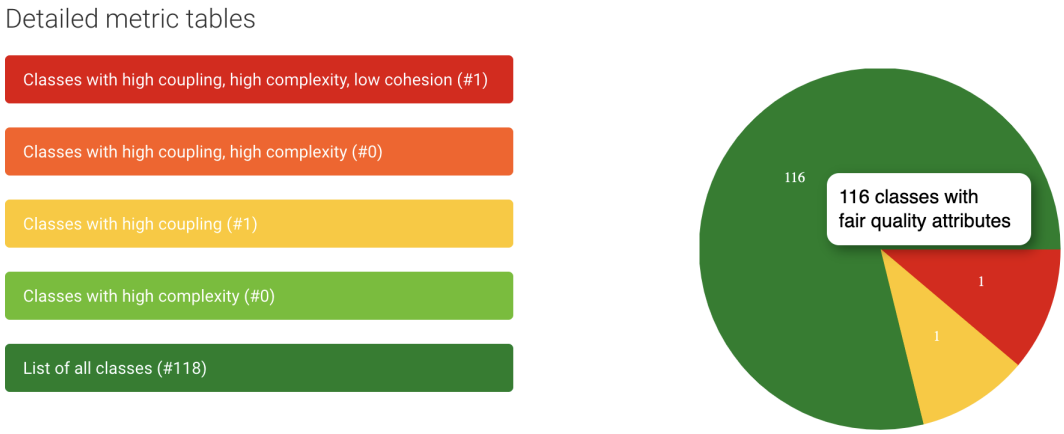

Figure 21. CodeMR Metrics Overview of CB-1

Among the analyzed classes, it is seen that two classes corresponding to 23% of the total code size have medium-high WMC values. It is seen that these classes, which are classified as A and B classes, have WMC values of 61 and 58, respectively. These classes are "Activities" extended from the "Activity" class provided by the Android SDK. There are also two other classes corresponding to 14% code size have low-medium WMC values. Rest of the classes appear to have low level of WMC values. When the values of the DIT metric of the classes are examined, it is seen that 14 classes, corresponding to 22.7% of the total code size, have middle-upper level DIT values and the other 14 classes corresponding to 26.4% of the total code size have low-medium level DIT values. When the NOC metric values of the classes are examined, it is seen that 7 classes corresponding to only 3.4% of the whole codebase have low-medium NOC values and all other classes have low NOC values. According to the results, it is seen that a class, which corresponds to 12.5% of the code volume of the application, has a very high COB value. It also appears that a class that is corresponding to 11.2% of the application's code volume has a high COB value, and another class that is corresponding to 4.4% of the code volume has a medium-high COB value. When the values of LCOM, the last of the selected metrics, are examined on a class basis, it is seen that 6 classes corresponding to 40.8% of the total code size have high LCOM values. Besides, it is seen that two classes have middle-upper level LCOM values, which corresponds to approximately 4% of the total code size. In the Fig. 22, the results of the class-based metric values, which is detailed and interpreted above, visualized by the "Metric Distribution" method offered by the CodeMR tool are presented. Charts are in doughnut form, and information regarding

the methods of creating them has been shared in the previous sections. As previously mentioned, the size/percentage of each part in the doughnut is proportional to the size of the class/interface it represents.

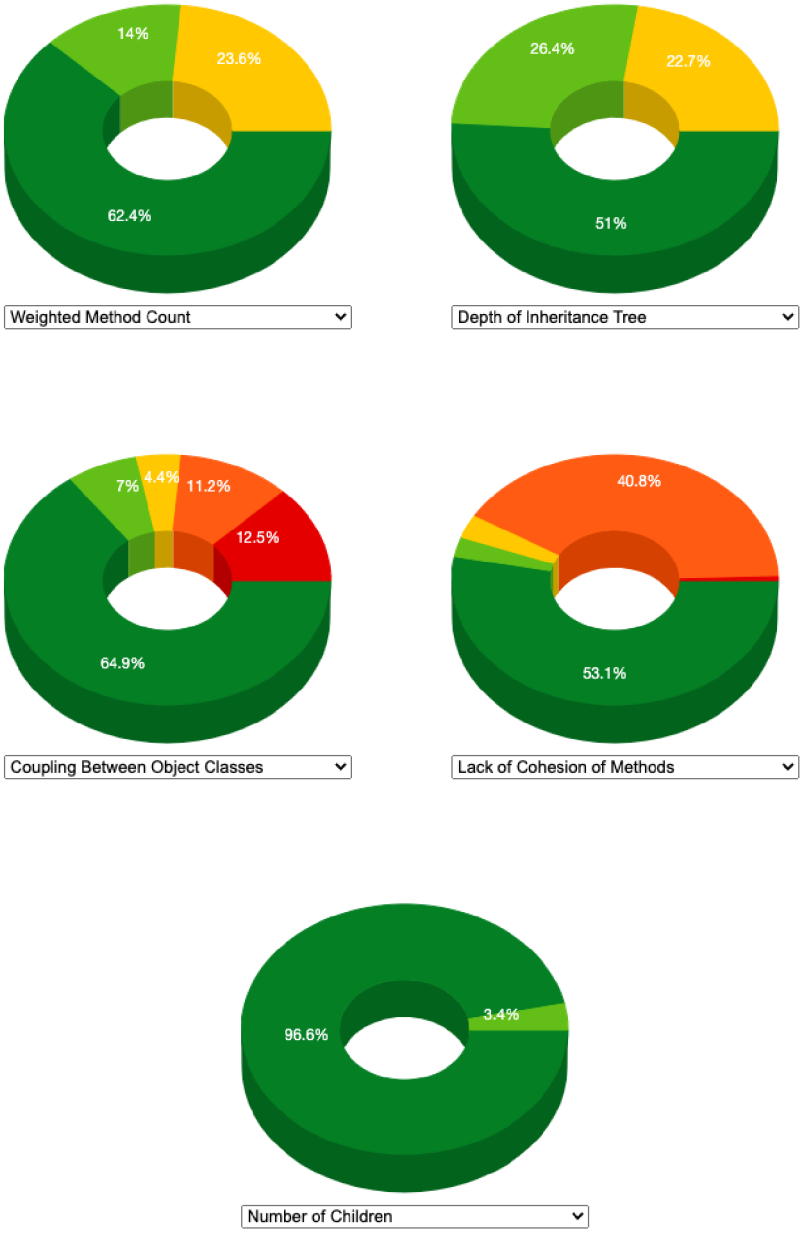

Figure 22. CodeMR Metric Distribution for CB-1

In the figure below, a general view of CB-1 in terms of complexity, coupling and cohesion is presented. In Fig. 23, the largest circle represents the project, while the inner

circles represent the packages and the small circles inside the inner circles represent the classes. The size of each circle is directly proportional to the size of the structure it represents.

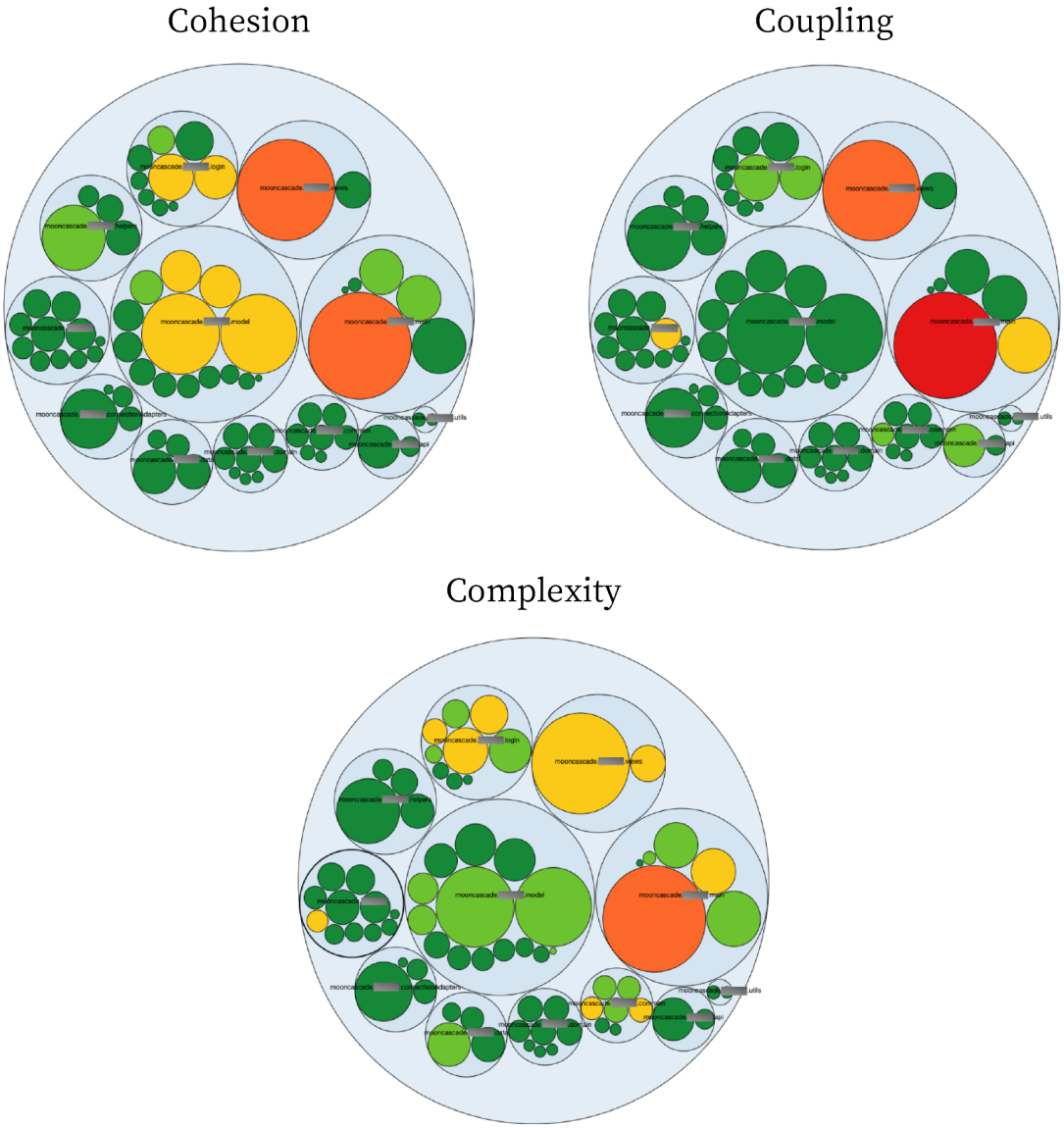

Figure 23. CodeMR Metric Values by Packages for CB-1

5.3.4 CB-2 Results

When the numerical analysis results produced on the CB-2 via the CodeMR tool are reviewed, it is seen that the tool analyzes 1160 lines of code belonging to this codebase.

These lines of code belong to 139 classes in 59 different packages of 4 different features which were mentioned in section 5.3.1. In Fig. 24, a CodeMR table that reflects the general situation of CB-2 is shared.

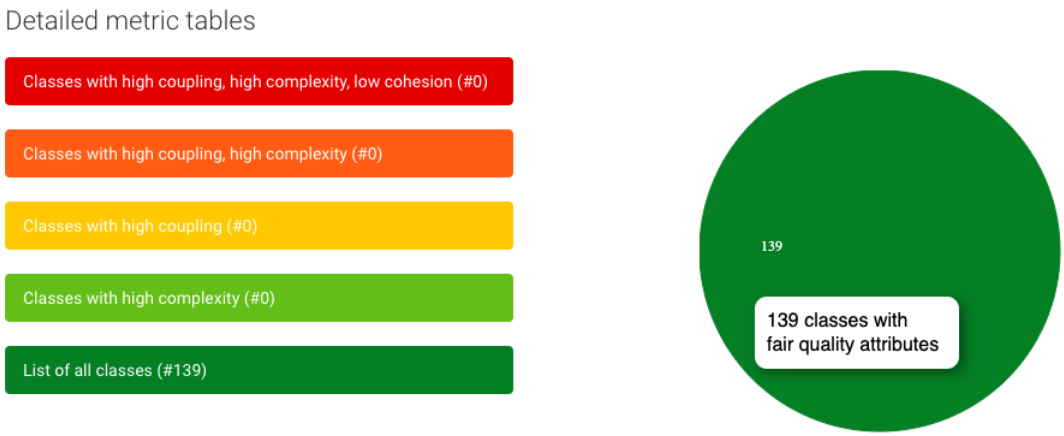

Figure 24. CodeMR Metrics Overview of CB-2

Between the analyzed classes, it is seen that two classes corresponding to 9% of the total code size have low-medium WMC values, and the rest of the classes appear to have a low level of WMC values. It appears that the majority of the classes belong to CB-2 have low complexity levels. When the values of the DIT metric of the classes are examined, it is seen that 10 classes, corresponding to 28.9% of the total code size, have middle-upper level DIT values and the other 27 classes corresponding to 14.1% of the total code size have low-medium level DIT values. The rest of the classes have a low level of DIT values. When DIT values of these classes are examined, it is seen that some classes have a DIT value of 2 and classes with more than a DIT value of 2 values are quite low (DIT values between 1-3 considered as low-medium by CodeMR). Classes with middle-upper-level DIT values are all view models or Activity/Fragment classes with a base class containing the common basic functionality. When the NOC metric values of the classes are examined, it is seen that 7 classes corresponding to only 6.6% of the whole codebase have low-medium NOC values, and 1 very concise class has medium-high NOC value. The rest of the classes have low NOC values. According to the results, it is seen that ten classes are corresponding to 26.6% of the application's code volume with a low-medium COB value and the rest of the classes have a low COB value. Results also showed that, all classes within the CB-2 have low LCOM values. Fig. 25 presents the results of the class-based metric values visualized by the "Metric Distribution" method offered by the CodeMR.

Figure 25. CodeMR Metric Distribution for CB-2

In Fig. 26, a general view of CB-2 in terms of complexity, coupling and cohesion is shown. As explained in the previous section, the largest circle represents the project, inner circles represent the packages and small circles inside the inner circles represent the classes. Sizes of the circles proportional to the size of the represented entity.

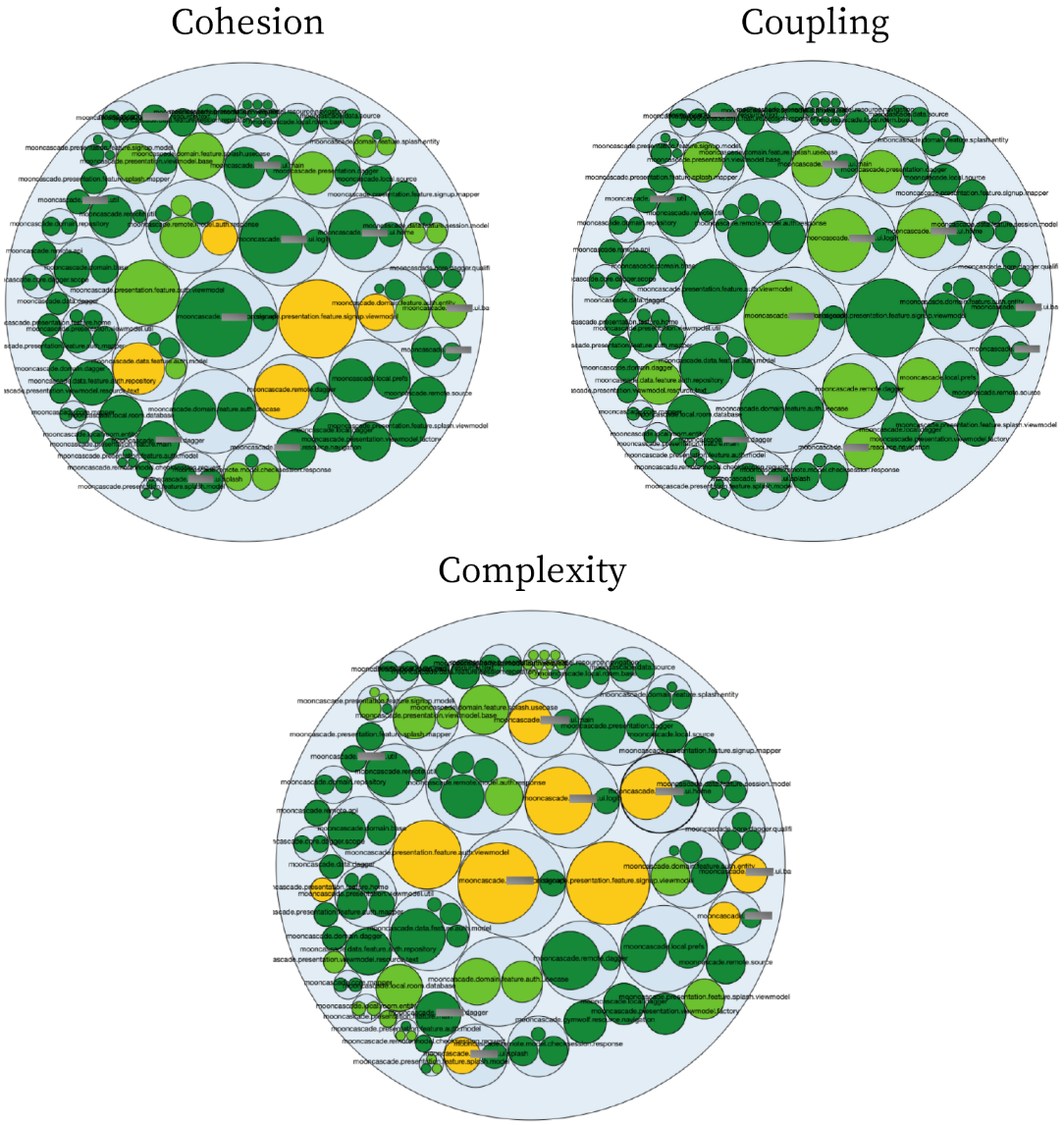

Figure 26. CodeMR Metric Distribution for CB-2

5.3.5 Feature-Based Comparison Results

The login feature has been selected among the available features to compare the maintainability differences between the two codebases. The reason for choosing it is that it has a more complex functionality/logic (e.g. validation, different error types, etc.) than other existing features mentioned in the previous sections. Classes containing view and view logic layers for the login feature from both codebases have been selected for comparison. The reason for selecting such layers is that these layers were considered the

most involved parts of the compared feature of the two codebases in terms of software complexity. While making the evaluation, other lower-level dependencies were excluded. When the CB-1 (uses the MVP design pattern) is examined, it is seen that 6 classes and interfaces are used related to the presentation and logic of data related to login. These classes and interfaces are: LoginActivityView (interface), LoginActivity (class), LoginActivityPresenter (class), LoginFragmentView (interface), LoginFragment (class), LoginFragmentPresenter (class). When the CB-2 (uses the MVVM design pattern) is examined, it is seen that only 2 classes are used and they are LoginActivity and LoginViewModel. The difference in the number of classes and interfaces used for the login feature in the codebases is due to the differences in the design patterns used and the fact that the CB-1 also uses the Fragment class of Android. Table. 2 presents the metric values obtained for each class from the quantitative evaluation performed using the Android Studio CodeMR plugin.

Class	Codebase	WMC	DIT	NOC	CBO	LCOM
LoginActivityView	cb-1	1	1	1	0	2
LoginActivity	cb-1	4	10	0	3	15
LoginActivityPresenter	cb-1	2	2	0	2	7
LoginFragmentView	cb-1	5	1	1	0	6
LoginFragment	cb-1	12	4	0	7	49
LoginPresenter	cb-1	14	2	0	8	43
LoginActivity	cb-2	5	10	0	7	48
LoginViewModel	cb-2	17	4	0	4	50

Table 2. CodeMR Metric Values for Login Feature

In order to examine the results better, groupings can be made between these classes, and the metric values of these groups can be compared. This grouping can be made based on the responsibilities of the classes. The responsibilities to be taken as a basis while determining groups are the view and view presentation. More detailed information on these responsibilities was shared in previous sections. These classes and their interfaces can be grouped for each codebase as follows. For CB-1, LoginActivityView, LoginActivity, LoginFragmentView and LoginFragment are only responsible for the view responsibility. LoginActivityPresenter and LoginFragmentPresenter are the classes responsible for how and when the data is presented. For CB-2, LoginActivity is responsible for the view responsibility, and LoginViewModel is responsible for how and when data is presented. This grouping will be useful for comparing the metric values shared in Table 2. Table 3

presents the metric values for each group of both codebases that are obtained from the analysis.

Codebase/Concern	WMC	DIT	NOC	CBO	LCOM
cb-1/view	22	16	2	10	72
cb-1/presentation	16	4	0	10	50
cb-2/view	5	10	0	7	48
cb-2/presentation	17	4	0	4	50

Table 3. CodeMR Metric Values for Login Feature

Results showed that the CB-1 codebase uses 6 entities for the selected layers of the login feature, while CB-2 uses only 2. From this point of view, the CB-2 has better organization and understandability. Moreover, when the values of WMC, DIT and NOC metrics used in complexity measurement are compared, it is seen that the complexity level of CB-2 for the view responsibility is lower. On the other hand, when the results of the complexity metric values of presentation related responsibilities are examined, it is observed that there is not much difference between the two code bases. These results should be considered normal given the complexity of functionality of the classes involved in this responsibility. When the CBO metric related to measuring the coupling level is examined, it is seen that the CB-2 codebase gives better results. Since the SOLID and DI principles are used much more effectively in the CB-2 codebase, these results are expected. Also, CB-1 uses the MVP design pattern. The coupling of the CB-1 is increasing due to the bi-directional dependency between view and presentation layers in the MVP design pattern. However, this situation is not the case for CB-2, which uses the MVVM design pattern. In the MVVM design pattern, only the view layer has a dependency on the presentation layer. When the results of the LCOM metric related to cohesion are examined, it is seen that the results are similar to the results of the complexity metrics. While the results of CB-2 are much better than CB-1 for the responsibility of view, there is not much difference between the results for the responsibility of presentation. There is only one view model per view principle in the MVVM design pattern. Therefore, one view model might have different responsibilities, especially those that belong to the complex views. The same is true for the MVP design pattern as well. There can be only one presenter for each view. Naturally, CB-1, which uses the MVP design pattern, seems to have low cohesion values for presentation responsibility.

6 Discussion

The interpretation of the data obtained as a result of answering the research questions within the scope of this study is presented in this section. Also, the effectiveness of the research method, the comments on the methods used by the case company that is mentioned in this study were shared. Besides limitations of the study is also discussed in this section.

6.1 RQ1 Discussion

First of all, the information obtained while answering the first research question guided the whole of this study. Research conducted to answer RQ1 has shown that the use of quantitative measurements together with qualitative measures can increase the effectiveness of the study. It was seen that qualitative measurements can make important contributions to the results of the evaluation with the information obtained directly from the developers and support the quantitative measurement results. The fact that experienced Android developers make evaluations about the methods and technologies they use every day from the maintainability point of view and the results obtained from these evaluations can be added to this study increased the study's accuracy. From this point of view, it would not be wrong to say that the addition of qualitative methods as well as quantitative methods to studies focusing on the measurement of software development concepts such as maintainability would increase the qualification of the study. The research conducted to answer the first research question showed that the maintainability of object-oriented software systems can be evaluated quantitatively by using many different metrics. In this study, a different quantitative maintainability model based on the concepts of complexity, coupling and cohesion was formed to measure the maintainability of Android applications. The formulation of this model is based on the problems encountered while developing Android applications mentioned in 1.1. Subsequently, five metrics that are suitable for this evaluation model and can measure these concepts were determined. However, it cannot be said that these methods and metrics used in this study is the best that can be used for this purpose. Although these methods and metrics were sufficient for this study, it would not be wrong to say that there may be more effective quantitative measurement methods. It would be appropriate to conduct comparative studies to determine the most effective solution.

6.2 RQ2 Discussion

First of all, when the results obtained from the Android developer survey are compared with the methods used by the Mooncasacade Android team, it is seen that the methods

used largely overlap with the community choices. There are slight differences (e.g. the preference of Kotlin Coroutines in the Android community, different design patterns) between the methods used by the team and the Android developer tendencies, but it can be said that these differences can be met normally. Although all the participants are Android developers, the users participating in the survey work in different companies and different domain, and it would not be wrong to say that the needs of these participants may vary according to the company and field they work in and that the methods and technologies they use are shaped according to their needs. On the other hand, the case of different user tendencies seen in the answers of the survey questions on the architectural pattern choice and the asynchronous event management libraries might be interpreted as the case company needs to re-evaluate the solutions used for these areas. Lastly, in several participants' responses, it was noticed that they choose based on the industry trends, and this situation is observed especially in the answers given by relatively less experienced developers. As stated in the previous chapters, the Android platform is a rapidly developing platform. Therefore, it is worth noting that while choosing the methods and technologies used for developing Android applications, decisions should be made based on long-term stability and maintainability rather than industrial trends.

When the results of the interviews with the case company's Android team were examined, it was seen that the team consists of highly experienced Android developers, and they are aware of the importance of maintainability in software engineering and Android applications. Therefore, it would not be wrong to mention that the quality of the answers given by the team members to the questions in the interviews is high. First of all, five out of seven participants stated that maintainability is vital for the company and Android applications due to the way the company operates and the nature of Android applications. The other two participants stated that maintainability is already essential for software systems. These results support the theory claimed in the problem statement section of this study that maintainability is a critical non-functional requirement for software systems, Android applications and the case company. Besides, while the participants' evaluations about the methods and technologies used by the company in developing Android applications are generally positive, some constructive criticism also draws attention. The necessity of reconsidering the choice of architectural pattern and the asynchronous event management libraries, which was also mentioned in the Android developer survey interpretations above, was also emphasized by some case company members in the interviews. This situation stands out as a topic that the case company should focus on. The difficulties of using the RxJava library and its beginning to become out of date and the usage of Clean Architecture causing too much complexity for small and medium-sized projects are indicated as the reasons for this situation by the participants. In addition, the participants expressed their views that making the methods used into more standard coding conventions and arranging these conventions as if they were a familiar language among all members of the team would have a positive effect

in terms of maintainability. Lastly, the views and awareness of most of the participants about choosing the libraries used in Android projects among stable and long-lasting libraries prove the effect of third party library use on the maintainability of Android applications.

Considering the quantitative evaluation results, some issues from CB-1 stand out. The first thing that catches the eye is a complex and unorganized packaging structure. Layer and feature-based packaging methods are internal in the project, which cause serious maintainability, understandability and organization problems. It is also noteworthy that some classes are large in size, which can be interpreted as lack of proper separation of concerns. The codebase appears to have complexity, coupling and cohesion problems as well. In a few classes, these problems are at a very high level, and maintainability and organization problems at different levels in the project draw attention. Especially the high coupling problem stands out for this code base. Considering that there are no healthy abstraction and dependency injection application in the project, this result is not very surprising. When all these analysis results are taken into account, it is clear that the CB-1 codebase shows a low maintainability character. Nevertheless, CB-1's situation offered an opportunity for this study to evaluate maintainability. On the other hand, results of CB-2 has shown that there are visible improvements in complexity, coupling and cohesion. In addition, significantly reduced entity sizes and increased class and package numbers were noticed. Concise classes and the increase in the number of packages point to a more organized code base and a better separation of concerns, making the understandability of the codebase high. Besides, the levels of the classes in complexity, coupling are quite low, and cohesion is high. This situation can be shown as proof that the SOLID and SoC principles are applied correctly, and it can be said that there is a very positive effect on maintainability. However, a few classes with moderate complexity and cohesion issues stand out. When these classes were investigated, it was seen that they were the classes called "View Model" in the MVVM design pattern that are responsible for how and when the data will be displayed. According to the principles of the MVVM design pattern, each view should have only one view model. In this case, the view models belonging to the views with more than one responsibility also have the logic of belonging to more than one responsibility. Therefore, these classes become more complex, and the cohesion of the classes decreases due to the methods and dependencies they have for different responsibilities. As long as the principles of the MVVM design pattern are followed, it would not be appropriate to divide these responsibilities between different classes. Nevertheless, the positive effects of the technology and principles used in the development of CB-2 on maintainability are obvious.

6.3 Limitations

First of all, it should be stated that the features used in quantitative evaluations were relatively less complex features of the Android application. Therefore the efficiency of evaluation and comparison was affected by this situation. Using more complex features could have provided better insight into the impact of the methods and technologies used by the case company. Unfortunately, this was not possible within the scope of this study due to not having the time required to develop more complex features. Still, it does not mean that these results are false or unrealistic. Moreover, considering that there may be insufficiencies in the methods used in the evaluations, the necessity to carry out more detailed studies to find a proper method for evaluating the maintainability of Android applications is undeniable. For example, there are different methods to evaluate the software system's maintainability, and different metrics can be used. Therefore choosing the most efficient metrics to measure the maintainability of software systems and Android applications is controversial, especially when the differences of Android applications are taken into account. Further research should be conducted to find the most efficient quantitative evaluation method.

7 Conclusion

This study covered the subject of evaluating the impact of the methods and technologies used by the software product development company Mooncascade while developing Android applications on the maintainability of these applications. There are four major challenges encountered when developing Android applications. These challenges are Android's nature, demanding business needs, the frequent update rate of Android applications, and changing development teams. A maintainability model was formed by focusing on these major challenges. This model was formed based on the correlation between the major Android development challenges and the well-known software engineering concepts such as complexity, coupling and cohesion, whose relationships with maintainability were proven. While applying this maintainability model, quantitative and qualitative evaluation methods have been used. Quantitative and qualitative evaluation methods were determined to measure maintainability and the evaluations were carried out using these methods. To make a qualitative evaluation, a public Android survey was carried out with random Android developers and interviews were conducted with each member of the case company's Android team. To make the quantitative evaluation, a set of object-oriented software metrics were used. These metrics were applied to common features of the different codebases belonging to the same project through the CodeMR static code analysis tool.

First of all, this study has demonstrated the importance of maintainability for Android applications and addressed the important matters in terms of the maintainability of Android applications. These matters stand out as usage of principles and conventions to increase software understandability, implementing human-readable code, proper software architecture and design pattern selection and use of stable third-party libraries. Quantitative and qualitative evaluations proved that the maintainability of Android applications developed by paying attention to these matters will increase. Moreover, results indicated the positive impact of the methods and technologies used by the case company on the maintainability of Android applications. The comparison between the two codebases, one developed using the case company's methods and technologies, the other developed without a specific order and standard, showed that maintainability increased even for the relatively simple application features. Outcomes also revealed the shortcomings and areas open to improvement for the methods and technologies used by the case company. The areas open to improvement are re-evaluating the use of libraries that are in danger of becoming outdated, such as RxJava, architectural scaling and selection according to the project, and making the coding conventions more standardized. It is predicted that re-evaluating these issues will further increase the positive effect on the maintainability of Android applications.

Lastly, the findings obtained as a result of answering the first research question

showed that a new model is needed to measure the maintainability of Android applications. The main reason for this need is the differences of Android applications from traditional software systems and their update rates. Especially Android's unique ecosystem points out the need for new methods and metrics to measure the maintainability of applications running on this ecosystem. While creating these new metrics, it is anticipated that besides the specific dynamics of Android applications, it may also be beneficial to use metrics that can include effort and time measurement that can be associated with high updating rates of the Android applications.

7.1 Future Work

Considering the lack of methods that can effectively measure the maintainability of Android applications, it would not be wrong to say that future research will focus on this issue as a continuation of this study. In addition, in the case of determining these methods, it is also among the targets to try the methods on more complex application features and get more effective results, thus eliminating the limitations of this study.

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] 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).

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

[4] Daniël Verloop. "Code Smells in the Mobile Applications Domain". In: *Master's thesis, University of Technology Delft* (2013).

[5] 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).

[6] 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).

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

[8] Shyam R. Chidamber and Chris F. Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering (Volume: 20, Issue: 6, Jun 1994)* (1994).

[9] I. Heitlager, T. Kuipers, and J. Visser. "A practical model for measuring maintainability". In: *In 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)* (2007).

[10] S. Easterbrook et al. *Selecting Empirical Methods for Software Engineering Research*. 2008.

[11] Oscar J. Romero and Sushma A. Akoju. "Adroitness: An Android-based Middleware for Fast Development of High-performance Apps". In: (2019).

[12] B. S. Panca, M.S. Mardiyanto, and B. Hendradjaya. "Evaluation of Software Design Pattern on Mobile Application Based Service Development Related to the Value of Maintainability and Modularity". In: *International Conference of Data and Software EngineeringAt: Denpasar, Bali* (2016).

[13] D.S. Kushwaha and A.K Misra. “A Complexity Measure Based on Information Contained in the Software”. In: *Proceedings of the 5th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems, Madrid, Spain, February 15-17, 2006 (pp187-195)* (2006).

[14] 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).

[15] 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].

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

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

[18] Ivano Malavolta et al. “How Maintainability Issues of Android Apps Evolve”. In: *2018 IEEE International Conference on Software Maintenance and Evolution* (2018).

[19] P. Bourque and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. 2008.

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

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

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

[23] Robert Cecil Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson Education, Inc., 2003.

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

[25] 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).

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

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

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

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

[30] David Garlan. "Software Architecture: a Roadmap". In: *ICSE '00 2000* (2000).

[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] Wei Sun, Haohui Chen, and Wen Yu. "The Exploration and Practice of MVVM Pattern on Android Platform". In: *4th International Conference on Machinery, Materials and Information Technology Applications (ICMMITA 2016)* (2016).

[33] Anastasia D. and Denys M. *Clean Architecture of Android Apps with Practical Examples*. 2018. URL: https://rubygarage.org/blog/clean-android-architecture. [Online; Accessed: 18.06.2020].

[34] S. Henry S. W. Li. "Object-Oriented Metrics that Predict Maintainability". In: *Journal of System Software, Vol. 23, pp. 111-112, 1993* (1993).

[35] Victor Oliveira, Leopoldo Teixeira, and Felipe Ebert. "On the Adoption of Kotlin on Android Development: A Triangulation Study". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2020).

[36] Lisa-Marie Andrä et al. "Maintainability Metrics for Android Applications in Kotlin: An Evaluation of Tools". In: *ESSE 2020: Proceedings of the 2020 European Symposium on Software Engineering* (2020).

[37] Henning Grimeland Koller. "Effects of clean code on understandability: An experiment and analysis". In: *Master's thesis, University of Oslo* (2016).

[38] Tung Bui Duy. "Reactive Programming and Clean Architecture in Android Development". In: *Bachelor thesis, Helsinki Metropolian University of Applied Sciences* (2017).

[39] 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).

[40] 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).

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

[42] 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).

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

[44] Rimmi Saini, Sanjay Kumar Dubey, and Ajay Rana. “Analytical Study of maintainability Models for Quality Evaluation”. In: *Indian Journal of Computer Science and Engineering (IJCSE)* (2011).

[45] 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).

[46] 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).

Appendix

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

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
14/05/2021

II. Attachments

The thesis is supplied with accompanying files.

Answer data set for the Android developer survey can be found on GitHub with the following URL: `https://github.com/tartarJR/Thesis/blob/master/Android%20Developer%20Survey%20-%20Responses.pdf`