**Abstract**

Recent decades have shown that people’s life has been remarkably affected by smartphones and mobile applications. This makes the development of Android applications one of the major areas in the software development industry. Nowadays, Android application development, which is based on one of the most successful open-source mobile operating systems, Android OS, has become one of the most necessary elements in the industry with a competitive and large market share. To survive in such a highly competitive market and frequent update rate, delivering Android applications of high quality is a must. The demand for high-quality Android applications comes from both users and developers. Consequently, developing a quality state-of-the-art Android app is very crucial for Android developers to minify the problems for the entire lifecycle of the app. Considering that developing such high-quality maintainable applications is a way to facilitate the issues that can encounter during the development lifecycle of an Android application, it is very vital to have the know-how for Android developers and researchers to develop high-quality Android applications that have the high level of maintainability. Consequently, this study seeks to provide thorough information regarding the fundamentals of building high quality and maintainable Android applications using Clean Architecture principles and state-of-the-art technologies. The study aims to provide detailed information regarding the best practices of developing Android applications using Clean Architecture. Furthermore, this study covers the evaluation of the impact of using Clean Architecture on maintainability when developing Android applications. The study focuses on native Android application development only.

**Key Words**

Android, Clean Architecture, MVVM, Dependency Injection, Architectural Components, Dagger 2, Reactive Programming, RxJava 3, Retrofit, Moshi, Kotlin

Yellow -> re-consider/re-write

Green-> reference

Orange-> vulnerable/do i need a reference/evidence

**1.Introduction**

In the last decade, the impact of smartphones on our lives has increased to a great extent, and smartphones and mobile applications became the 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. Notably, Android, as a successful open-source mobile operating system, has been a core element of this change, and the demand for Android applications has increased. Nowadays, Android application development has become one of the most necessary parts of the business area with a significant market share. Today, there are more than 2.5 billion active Android devices in the world(1). Therefore, it is difficult to ignore the importance of Android application development, considering the market share of the phones operating with the Android operating system and a large number of people's interactions with mobile applications running on Android devices. Consequently, Android application development has become an essential topic in the IT industry and academy. However, the increasing importance of the mobile era also brought more challenges to mobile application development and, of course, in particular for Android application development. Today, the difficulties that arise during Android application development can be examined under four major topics.

Firstly, Android applications are distinguished from traditional web and desktop software with their sophisticated features and specific structure. A typical Android app consists of components such as views, activities, fragments, services, broadcast receivers, and content providers. These components are unique to the Android platform and provided by the Android SDK. The Android system mostly controls the behaviors of these components. Android developers must obey to contract supplied by the system when using these components, and this situation sometimes limits developers to use some specific programming techniques when developing Android applications. Besides, there are other components used in an Android app that are not a part of the Android SDK for managing network operations, database transactions and, business rules, and so on. Each of these components, whether it is a part of the Android SDK or not, represents a different "concern" of an Android application. The necessity of working together in harmony for all these components, the challenges that arise from the Android operating system and Android SDK's nature, and the limited resources of mobile devices make Android applications complicated software systems that are hard to develop.

Secondly, Android applications get more and more sophisticated to fulfill increasing user needs and business requirements. Mobile applications become more functional as user and business needs increase. Consequently, the complexity of Android apps from the software development point of view increases, and apps become more business-critical.

Thirdly, Android applications have a high update rate because of bug fixes and the frequent addition of new features based on the changing business requirement and user needs. For that reason, Android applications should be developed in a way that the addition of new features and fixing bugs can be done smoothly.

Last but not least, it gets harder to maintain the codebase, as the codebase and the development team grows or changes. The codebase of an Android application has to be in an orderly fashion that enables developers to read and understand the purpose of the app quickly. Also, any time a new developer joins the team, the time required to onboard a new developer to the codebase is directly related to the way that the Android applications were developed.

Therefore, developing high-quality Android applications to overcome the above-mentioned challenges in a time and cost-efficient manner in terms of software engineering principles and software quality standards is essential. Hence the critical question is how to achieve such a goal when developing Android applications. Over the last decade, a couple of different ideas have been in place to resolve these issues in the context of Android application development, and similar views with the same purpose are continually evolving. However, the primary purpose of all these ideas is the same. All these ideas and methodologies, whether in the form of software architecture, design patterns, or coding conventions, aim to improve the "maintainability" of the Android applications and to apply the principle of "separation of concerns" properly to the Android apps.

In the light of the above considerations, the primary goal of this study is to identify, study, and share the methodologies and technologies used by Mooncascade, a leading software product development company with global reach, for developing real-world enterprise Android applications. These approaches used by Mooncascade aim to provide Android apps with high quality, maintainability, and proper application of separation of concerns. By using these methods and technologies, it is intended to solve the difficulties mentioned before. The study also analyses the impact of the methodologies and technologies from the maintainability point of view. To that aim, both qualitative analyses (in the forms of interviews, questionnaires) and quantitative analysis (in the way of measuring maintainability metrics) will be conducted.

Since the study topic is highly coupled to industry and industry trends, the author contributes to identifying and understanding the practices used by one of the top software development companies in the region and their impact and bringing those practices into the academy through this study. The target audience of this study includes Android developers and researchers who are already experienced with Android application development basics and willing to learn advanced techniques and tools for Android application development. The study facilitates developers and researchers to follow the most up-to-date software engineering and Android development practices to develop state-of-the-art Android applications with high maintainability. The study only focuses on native Android application development.

**1.1 Scope and Goal**

This study addresses the practices used by Mooncascade to resolve the challenges mentioned in the introduction section. To this end, the study aims to present comprehensive and up-to-date resources used by Mooncascade's Android team, including the tools, libraries, and techniques and how those are used to achieve the goal.

The study will identify, understand, and share the practices and technologies used by Mooncascade's Android team. Moreover, it will present the determined practices in the forms of code samples and instructions. In this way, the study aims to facilitate resolving the challenges mentioned in the introduction section and provide advanced techniques for developers and researchers when developing state-of-the-art Android applications.

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

**1.2 Contributions**

The main contribution of this study is providing comprehensive information regarding the development of large scale and enterprise Android applications through the experience of a proficient software development company, Mooncascade. With the help of the information that this study offers, developers and researchers will be able to know the fundamental principles and technologies required to develop state-of-the-art Android apps that have high maintainability where the separation of concerns applied decently. Thus solving the difficulties explained in the introduction section can be facilitated.

Examination of white and gray literature has shown that there there is not a similar case study regarding Android application development. Still, some studies focus on maintainability and the quality of Android applications and also solutions for challenges mentioned earlier in this study. However, the examination has also shown that such studies lack detail and up-to-dateness. The valuable set of information that this study offers might help to fill the gap of outdatedness and lack of detail between the industry and academia in developing Android applications. Unlike most similar studies, in this study, the investigated principles and technologies will be gathered from the real-life Android application development best practices of the industry, and the study will include detailed information about the implementation of these principles and technologies. Apart from the main contribution, the evaluation results of presented methodologies and technologies will provide empirical data, both from the maintainability point of view and the Android developer's point of view. Lastly, the Android developer survey conducted as a part of this study can provide insights from the latest technology trends in the Android community.

**1.3 Thesis Outline**

The rest of this study is structured as follows:

**2. Background**

This section starts with some basic information about the Android environment and the nature of the Android applications. With this information, it is aimed to facilitate understanding of how the nature of Android affects the maintainability of Android apps and why the same nature makes the adaptation of the principle "separation of concerns" important when developing Android applications.

Later, the section continues with some programming and software engineering fundamentals. First, it describes "maintainability" and "separation of concerns" from the software engineering point of view. Later, it explains why these terms are essential for software engineering and then mainly in Android application development. Besides, the information regarding the issues caused by the lack of maintainability and poorly applied separation of concerns will be discussed. After that, the section explains what software architecture is and what relation it has with the above-mentioned concepts. In addition, software architecture in general in the context of Android will be covered.

Lastly, an overview of some popular solutions from industry will be shared. Lastly, an overview of the white and gray literature review conducted as a part of this study.

**3. Problem Statement**

This section explains the problem that is addressed in this study. A detailed explanation of the problem, along with the reasons behind the problem, will be presented in this section. Also, research questions will be defined and explained in this section.

**4. Mooncascade Case Study**

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

**5. Evaluation**

The impact of the practices that are used by Mooncascade's Android team to the maintainability of Android applications will be evaluated in this section. The evaluation will be both in qualitative and quantitive form, and the details of the evaluation methods will be explained as well. In order to fulfill the evaluation in the qualitative form, surveys will be conducted amongst the Mooncascade's Android team members. The quantitive evaluation will be fulfilled by using some software system maintainability metrics. These metrics will be applied to two different versions of the same Android application. The practices identified by this study will be applied to one of these versions and then compared to the old version of the same application, which is poorly structured. The metrics will be applied to the same use cases from these different versions in order to get the best evaluation results. Lastly, in order to identify the accuracy of the practices used by Mooncascade's Android team, a developer survey will be conducted amongst the Android community, and the results will be shared.

**6. Discussion**

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

**7. Conclusion**

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

**2.Background**

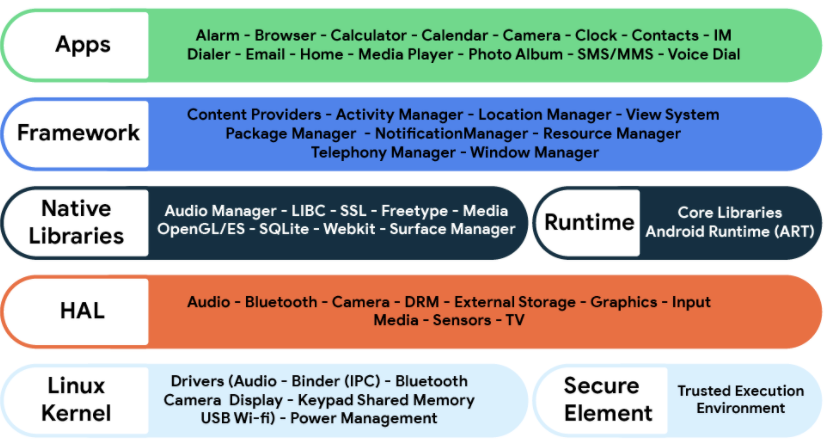
***“The only way to go fast is to go well.” - Robert C. Martin (11)***

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

**2.1 Android**

Android is an open-source operating system for mobile devices. The Android project/operating system was initially created by the Open Handset Alliance which includes organizations from various industries such as Google, Vodafone, T-Mobile, LG, Huawei, Asus, Acer, and eBay to give some examples (6). To be more specific, Android is an open-source software stack made for a varied range of mobile devices with different structure parameters. The main goal of the Android project is to provide an open software platform accessible for a variety of stakeholders such as developers, engineers, carriers, and device manufacturers to turn their innovative and imaginative ideas into successful real-world products that improve the mobile experience for the end-users. Today, numerous organizations from Open Handset Alliance and also other organizations are supporting and investing in Android and the project is led by Google. Android is designed in a distributed way to avoid the issue of the central point of failure. In another means, different industry players confine or control the advancements of another. As a result, a production-quality consumer product comes along with open source code that is ready for customization(5).

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



**Figure 1.** Android platform architecture (5)

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

**2.1.1. Applications**

The application layer is the top layer of the Android platform architecture. The Android operating system provides a decent number of built-in applications for fundamental user needs such as contacts, calling, SMS messaging, web browsing, email, SMS, calendars, etc. and more (7). These built-in applications are developed by the Android team. In addition to these built-in applications, there are also third-party applications that are developed by third-party providers. All these applications are installed in the application layer. Android provides the flexibility for developers to write applications that can replace any built-in application provided by the Android operating system. This study focuses on the best practices to develop the maintainable and testable state of the art Android applications that will be installed in the application layer.

**2.1.2. Java API Framework**

The Android operating system's APIs written in Java programming language are provided through this layer. These APIs are the core building blocks for the developers in order to develop Android applications. A couple of noticeable key core building blocks include the following (7):

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

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

**2.1.3 Native Libraries**

This layer includes a variety of C/C++ core libraries and Java libraries. The purpose of this layer is to provide support for core features. Important system components of Android, such as HAL (Hardware Abstraction Layer) and ART (Android Runtime) depend heavily on the native libraries that are written on C/C++ programming languages(7). Some of the important native libraries are Camera, Media, OpenGL, SQLite, WebKit, SurfaceManager, etc.

**2.1.4 Android Runtime**

Android Runtime(ART) is a runtime environment for applications that run on the Android operating system. With the launch of Android Version 5.0 or in other words, API level 21, the Dalvik virtual machine was replaced with Android Runtime. Since then, every application started running on its own process along with its own instance of Android Runtime (7).

**2.1.5 Hardware Abstraction Layer**

Hardware Abstraction Layer(HAL) is responsible for providing the device hardware features to the Java API Framework layer through interfaces(7). In other words, it enables the communication between the device hardware and framework. The hardware includes important device features such as Bluetooth, camera, sensors.

**2.1.6 Linux Kernel**

The very core of the Android platform is the Linux Kernel and the base infrastructure of the Android heavily depends on the Linux. Using Linux comes with some advantages. Firstly, using a well-known kernel enables device providers to work on a platform that they already recognize, and second Android benefits from the Linux system and kernel security (7). This layer also provides all the hardware drivers for camera, display, etc and handles battery management and other system-related properties.

**2.2 Android Applications**

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

Ever since the Android operating system has started running on mobile devices the Android applications are being developed. As of the first quarter of 2020, there are more than two and a half million applications in the Google Play Store(8). Since the launch of the first mobile device that works with Android, the Android operating system has improved and the way the Android applications are developed changed a lot but some fundamental components for developing Android applications have more or less stayed the same. Each of these fundamental components was developed for a specific purpose by the creators of the Android Software Development Kit(Android SDK). In order to understand the problem that this study is trying to solve, it is very important to know what these components are and what their responsibilities are. Because as it was already stated in the introduction section, oftentimes the complications and complexities that arise when developing Android applications come from the nature of these components and the necessity of working together in harmony for all these Android components. So, in this section, some brief information will be given about these fundamental Android components. For more information and technical details regarding these components, it is recommended to read official Android documentation.

**2.2.1 Android Application Fundamentals**

Java, Kotlin, and C++ programming languages can be used for developing native Android applications(9). There are also other ways of developing Android applications. But as it was already mentioned in the introduction section, this study only focuses on native Android application development. Native Android development means the creation of Android applications that run on Android-powered devices by using the Android Software Development Kit.

When developing Android applications in a native way, in addition to the programming side, Android applications are supported by different types of resources such as XML layout files, XML resources, images, data files, etc. The detailed examination of these resources is not within the scope of this study. However, knowing that Android applications do not only consist of code might be useful to see the bigger picture of an Android application. Though, for the purpose of understanding the problem that this study tries to resolve, it is essential to have a basic understanding of the fundamental Android components. Consequently, knowing the nature of an Android application and its components is the first step for solving the maintainability issues of the Android applications and then there come the best practices and latest technologies of Android application development and how to apply them into the Android application development processes. These fundamental components can be listed as(9):

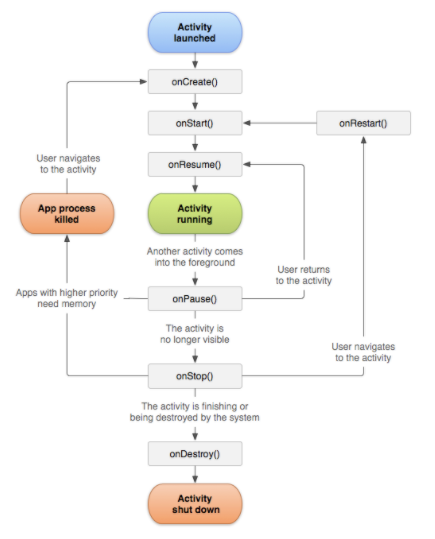
* Activities
* Services
* Broadcast receivers
* Content providers

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

**2.2.1.1 Activity**

In the Android environment, the term “activity” refers to the interaction entry point of Android applications for the end-user. Alternatively stated, an activity is a screen with a user interface. It would not be wrong if we say that activity is the main component for building an Android application. Activities are represented by the “Activity” class of the Android Software Development Kit. Each activity of an Android application is implemented as a subclass of the Activity class(9). An Android application might have single or more activity. Oftentimes an Android application consists of a set of activities. An activity can start and finish another activity which means activities are also responsible for the navigation within the application. An activity can even start another Android application from the device. Activities within an app provide a strong user experience together. Since the activities of an Android application can affect each other and each activity has its own lifecycle, this situation has an important impact on the maintainability of the Android applications. Thus activity lifecycle should definitely be taken into account when developing Android applications. During the lifetime of activities, some actions should be taken and some methods should be called depending on the situation. Some of the important ones of these situations are covered in the upcoming sections but before discussing these, it is important to understand the activity lifecycle.

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



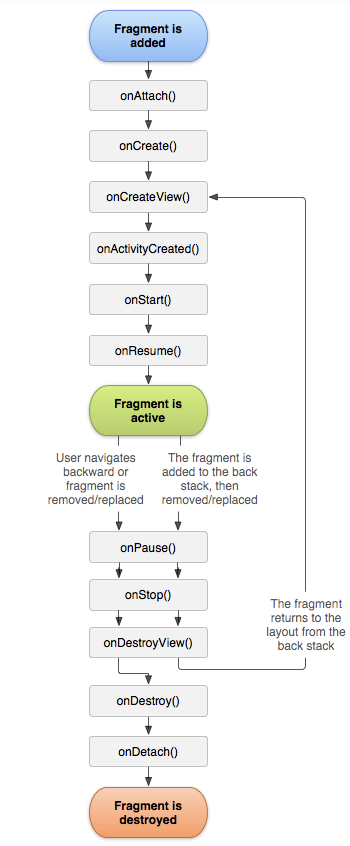
**Figure 2.** Android activity lifecycle (10)

Improved knowledge of activity lifecycle callback methods is not only helpful for maintaining system resources, organizing the activity behaviors during the state changes, and preventing undesirable situations. Noteworthy that the knowledge of activity lifecycle callback methods is also useful when it comes to integrating non-UI related components of an Android application to the activities so that activities and these other components can be used together in harmony with high efficiency. Good knowledge of these callback methods enables developers to be aware of when to add/release these non-UI related components to activities when building maintainable Android applications. A couple of examples from such situations are covered in the upcoming parts of this study. In order to make the illustration above more meaningful and understanding the role of each activity lifecycle callback method, the purpose of each activity cycle method is briefly mentioned below(10).

* **onCreate:** This callback method must be implemented by every activity. The onCreate callback method is invoked by the system when the Android system creates the activity. Following the creation of the activity, the activity goes into the "created" state. The method can be used to set the user interface layout, bind initial data to view elements, creating the instances of the activity scoped variables. The method can also be called as a result of a configuration change. The activity does not dwell in the created state. Following the execution of the onCreate method, it immediately enters the “started” state.
* **onStart:** This callback method is invoked by the system when the activity gets in the “started” state. With the invocation of the callback method, the activity becomes visible to the user and the app organizes the activity to move to the foreground and become interactive for the user.
* **onResume:** This callback method is invoked by the system when the activity gets in the “resumed” state and moves into the foreground. The resumed state is the state that users can interact with the Android application. An activity stays in the resumed state unless something else changes the focus of the Android device (e.g., navigating to another activity or application). When the focus of a device changes, the activity enters the “paused” state with the invocation of the onPause callback method. Then if the activity comes back to the foreground and returns to the resumed state, the onResume method is called again by the system. Consequently, the onResume callback method is the place that initializes components to be released in onPause callback method.
* **onPause:** This callback method is called by the system as an initial signal that the activity is being left by the user however that signal does not mean the activity that is left has to be destroyed by the system. The invocation of this callback just indicates that the activity which the onPause method is called is no longer in the foreground. The onPause callback method can be used to release system resources and stop operations that do not need to run when the activity is in the foreground. Since the execution time of the onPause callback method is very short, developers should not use the method to perform time-consuming operations such as performing database transactions, making network calls. The execution of the onPause method does not mean that the paused state of the activity changes. It remains the same until the activity goes back to the resumed state or the activity becomes completely invisible to the user. It is important to know that a paused activity might still be visible in multi-window mode. For that reason, the onStop callback method should be used instead of the onPause to release UI related resources.
* **onStop:** The onStop callback method is called by the system when the activity gets into the “stopped” state and no longer visible to the user. This callback method can be invoked in situations such as when another activity comes to the foreground or when an activity is about to be terminated by the system. The onStop callback method should be used to release and regulate the resources that are not needed when the activity is no longer visible to the user. The onStop method is also the correct activity lifecycle method to perform operations for shutting down the CPU-heavy processes. From the stopped state, the activity either goes back to the resumed state and starts interacting with the user again or the activity is finished. In the former case, the system calls onRestart, onStart, and onResume callback methods respectively. In the latter case, the onDestroy callback method is invoked by the system.
* **onDestroy:** This callback method is invoked by the system before the activity is destroyed. The reason for the invocation of this callback method is either the activity is finishing (e.g., the user dismisses the activity or the finish method is called) or the activity is being destroyed by the system due to a configuration change or low memory situation. All the resources that are not released in the earlier callback methods have to be released in the onDestroy callback method. This callback method is the last lifecycle method that the activity receives.
* **onRestart:** This method is called by the system when the activity is being re-displayed to the user after it was stopped. The invocation of the onRestart method is followed by the invocation of onStart and onResume respectively(12).

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

A fragment can be considered as a modular part of an activity. Fragments can be combined in an activity to build multi-window user interfaces. Fragments have their own lifecycle and input events. Unlike activities, fragments are reusable and reuse of a fragment in multiple activities is possible. Fragments cannot exist on their own. Every fragment is hosted by an activity. That makes the lifecycle of a fragment directly related to the lifecycle of the host activity that the fragment was created in. Fragments are represented by the “Fragment” class of the Android Software Development Kit. In order to create a fragment, a class must inherit the Fragment class or existing subclasses of the Fragment class(13). It was already indicated that, though fragments have a lifecycle that is tightly coupled with their host activity lifecycle, they have their own lifecycle which is a bi different than the activities’ lifecycle. Thus fragments have their own lifecycle callback methods. The study was already aforementioned that why having good knowledge regarding activity lifecycle callback methods is important. The importance of the fragment lifecycle callback methods is no different as well. Proper understanding of fragment lifecycle callback methods helps developers to know when to add/release these non-UI related components to fragments when building maintainable Android applications based on fragments. The figure below is an illustration of the fragment lifecycle and it presents the fragment lifecycle methods.



**Figure 3.** Android fragment lifecycle (13)

Just like the activity lifecycle, the fragment lifecycle exists in three states as well and these states are "resumed", "paused", and "stopped". Transitions between these states are managed by the system through the callback methods presented in the figure above(13). Below are the brief explanations of each fragment lifecycle callback methods(14):

* **onAttach:** This callback method is invoked by the system when the fragment has been linked with its host activity.
* **onCreate:** This callback method is invoked by the system when the fragment is being created. The initiation of the components needed by the fragment should be made in this method.
* **onCreateView:** This callback method is invoked by the system in order to create the view hierarchy of the fragment.
* **onActivityCreated:** This callback method is invoked by the system when the host activity's onCreate method has finished its execution.
* **onStart:** This callback method is invoked by the system when the fragment is visible.
* **onResume:** This callback method is invoked by the system when the fragment is visible and ready for interaction with the user. The fragment goes to the resumed state only after its host activity goes to the resumed state.
* **onPause:** This method is invoked by the system as an initial indication that the fragment is being left by the user. Following the invocation of this callback method, the user will not be able to interact with the fragment. The indication does not mean that the fragment will immediately be destroyed. The user might come back to the fragment at some point and that is why this callback method is the place where developers should save any changes.
* **onStop:** This callback method is invoked by the system when the fragment is no longer visible to the user
* **onDestroyView:** This callback method is called by the system when the view hierarchy of the fragment is being destroyed. The invocation of the onDestroy method is not guaranteed by the system. Therefore, any resources or views within the scope of the fragment should be released in the onDestroyView method.
* **onDestroy:** This method is called by the system when the fragment is no longer used by the user. Its purpose is to do the final clean up of the state of the fragment.
* **onDetach:** This callback method is invoked by the system when the fragment is being detached from the host activity.

**2.2.1.2 Service**

In the Android world, the term "service" refers to a component that runs in the background in order to perform time-consuming and long-running processes or to perform remote processes. In other words, a service is an entry point for Android applications to keep the applications running in the background for any reason(12). A service does not have a user interface. Other Android components such as activities, fragments can start a service. Once a service is started, it continues to its long-run even if the user starts another application. Android services can be used to perform background operations such as network operations, content provider interaction, I/O processes, playing music, etc.(15). In Android, a service is represented by the "Service" class and every service must be implemented as a subclass of the Service class that the Android SDK provides(12).

**2.2.1.3 Content Providers**

In the Android world, the term "content provider" refers to the component that is designed to manage a mutual application data set that can be stored via a file system or a local database(e.g. SQLite) or any other kind of persistent storage. Content providers define, manage, and supply inter-application data sets. Android applications can provide content providers for other Android applications and through the content providers, any other Android application that has the necessary permissions can query the content provider to read and write data within its permissions. From the Android system perspective, a content provider can be considered as an entry point into an Android application in order to issue named data sets identified by a URI scheme. A solid example to a content provider can be given as the content provider that the Android system provides for the purpose of managing the contact information of the user between multiple apps(12).

**2.2.1.4 Broadcast Receivers**

The term "broadcast receiver" refers to the Android system component that enables the system to distribute events that happen outside of the normal application flow to the Android applications. Just like the other main Android application components that were mentioned previously, broadcast receivers are also entry points to the Android applications. As a result of that, the Android system can deliver broadcasts to Android applications regardless of the application is running or not. Broadcast in Android tends to originate from the Android system itself. Low battery notification, captured screen notification, the screen on/of indicators can be given as examples to the Android system broadcasts. However Android applications can also initiate broadcasts. Broadcast receivers do not involve displaying a user interface but they have the ability to create notifications in the status bar in order to alert users. Android Software Development Kit provides the "BroadcastReceiver" class and each broadcast receiver must be implemented as a subclass of this provided class in Android applications(12).

**2.3 Maintainability**

***"Good programmers write code that humans can understand." - Martin Fowler(29)***

In ancient times, when computers were big, heavy, and slow, programmers were limited to use low-level programming languages that are working close to computer CPUs. These were imperative programming languages, and the programs written in these programming languages were following the procedural programming paradigm. Although that approach works fine, the biggest problem was that these programming languages were designed to be understood by computers, not humans. The main reason for this situation was that, back in that time, computers lacked proper hardware, resources, and speed. Consequently, the priorities back then were different. The computer programs had to be fast and less memory consuming. However, this situation has changed in the current day. Even a low-quality mobile device is much stronger and smarter than the computers that we were using a couple of decades ago, and software systems became complex. Although this change brought a positive impact on the end-user side as it also brought more functionality and ease, the impact it brought the software development side is complexity. Especially when developing large-enterprise software products, ignoring that fact and not considering how to overcome this complexity may cause significant failures. In this context, new programming languages and paradigms were born, the priorities have altered, and this new reality brought different challenges and new quality standards to software development(17).

When the priorities of modern software development are analyzed today, the maintainability of software systems emerges as one of the most critical priorities, perhaps even the most important. According to the IEEE Standard Glossary of Software Engineering Terminology, the term "maintainability" is the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed(28). The level of maintainability that a software system depends on several different factors. Overall, a software system can be considered as maintainable if it is simple to grasp how it works and what it does, and making changes such as adding new features and fixing bugs is easy. 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(31).

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

The importance of maintainability for software systems can also easily be seen when looking at its role in the software development lifecycle and its effect on software development costs. The time has to be spent on the maintenance of complex software products is comparatively more extended than the rest of the software development lifecycle processes. Reports indicate that the amount of effort spent on software maintenance is between 65% and 75% of the total amount of effort(18). Also, another report points out that maintenance cost is 75% of the total project cost, and the cost for maintaining source code is ten times bigger than developing the source code(30). In his famous book "Clean Code," Robert C. Martin explains how a top-rated company in the late 80s was wiped out from the business due to the lack of maintainability and poorly managed code organization. When the release cycles of their prominent product extended, due to the unorganized code base of their product, they were not able to fix bugs, prevent crashes, and add new features. Eventually, they had to withdraw their promising product from the market and went out of the business. Lousy code and consequently, maintainability was the reason for this company to go out of the business(16). This real-life example clearly shows how vital maintainability is to software systems and what fatal consequences it can cause if ignored.

Android application development is no exception for considering maintainability as a critical aspect of the software development lifecycle. In fact, the importance of maintainability for Android applications is even higher since Android applications have a very active software development life cycle. Given that the growing user demands and business needs making the Android applications more and more complex and Android applications having frequent update rates, it is not hard to see that how important the maintainability is for Android application development.

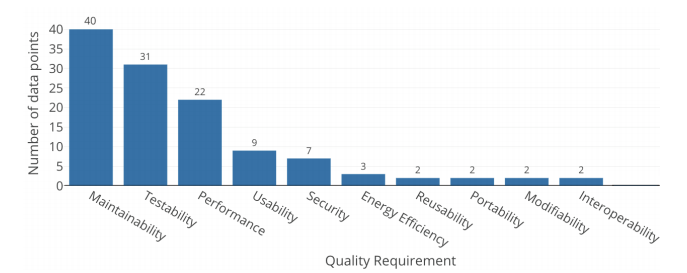
**2.4 Software Architecture**

Software systems are not static environments. They grow, change, and get updated based on the new requirements. During this change, new additions are made in forms of components, methods, modules, etc. As the system gets bigger the interactions between these different system elements also get more complicated and all together lead to complexity in software systems. In order to be able to develop reliable software systems that can overcome this complexity, programmers must implement software systems in a generalized fashion. Software systems that are written in such fashion can be considered reliably usable, maintainable, testable, and extendable(20).

At that stage, the question is what would be the so-called generalized fashion from the software engineering perspective. The answer to the question is software architecture. Martin Fowler defines the meaning of the architecture in the software industry as "the shared understanding that the expert developers have of the system design"(22). More technically speaking, software architecture is "the set of significant decisions about the organization of a software system, the selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these elements into progressively larger subsystems, and the architectural style that guides this organization -- these elements and their interfaces, their collaborations, and their composition"(23).

When contemplating the same situation for Android application development, of course, there is no difference. In fact, when the fast update rate of Android applications, frequent requirement changes of Android projects, and the complex issues and restrictions that arise from the nature of Android are taken into consideration, the significance of software architecture selection when developing Android applications becomes even more prominent. This circumstance is the source of motivation for this study.

And again at this stage, another question arises: With many options are available architecture Android applications, what is the top quality requirement that an Android application architecture should provide in order to overcome the complexities? As it has been constantly emphasized by this study since the beginning, the answer to that question is maintainability. A related study conducted between Android practitioners and other related academic papers reveals that the top quality requirement for architecting Android applications is maintainability(19).



**Figure 3.** Quality requirement rankings for architecting Android apps (19)

It is proven that maintainability and selection of architectural patterns when developing Android applications are directly in correlation. As a consequence, the impact of the architectural pattern on the maintainability of Android applications is one of the top aspects that should be considered when making the decision of the architectural pattern before starting the development of a new Android application.

**2.5 Existing Solutions and Literature Review**

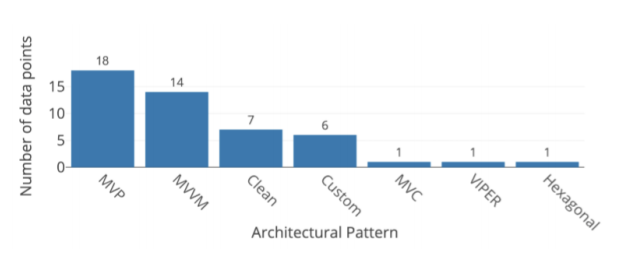
In the previous section, it was clearly stated the importance of maintainability and consequently the significance of architectural patterns. Moreover, it was indicated why the maintainability and architectural pattern selection is even more important for Android application development. However, that was not always the case for Android applications in the earlier periods of the Android application development. During the early days of Android development, Android developers tended to use "god activities". God activities were activities where developers place all kinds of code including user interface, presentation logic, business logic, and so on. This unstructured way of a software development approach that does not follow any architectural or design pattern is called "anti-pattern". Back in those times, Android applications were relatively smaller and less complex. Ever since the Android applications started becoming more and more complex these god activities became a black hole for Android developers because they were hard to read, understand, test, and maintain. Android developers were quick to recognize these problems and needs and had no trouble finding solutions on how to develop Android applications in a more organized way.

However, this area has been problematic for Android application development and Android developers from the beginning. With improvements and advancements in hardware and software and growing demand for the user and business needs, the requirement for developing organized and maintainable Android applications has had a great extent(25). In order to solve this god activity and such an anti-pattern problem and improve the maintainability of Android applications, Android developers started applying well-known and widely used architectural and design patterns in GUI-heavy applications to Android application development(26). Although the use of architectural patterns and design patterns in Android application development does not seem to be a requirement today, it is actually of a de-facto must.

Today, indeed, developing an Android application without applying an architectural pattern would be a bad decision because complex applications that do not follow any architectural pattern are expected to end up with serious maintainability issues. Also surviving in such a competitive market is heavily dependent on developing a well-architected Android application that has a high level of maintainability(19).

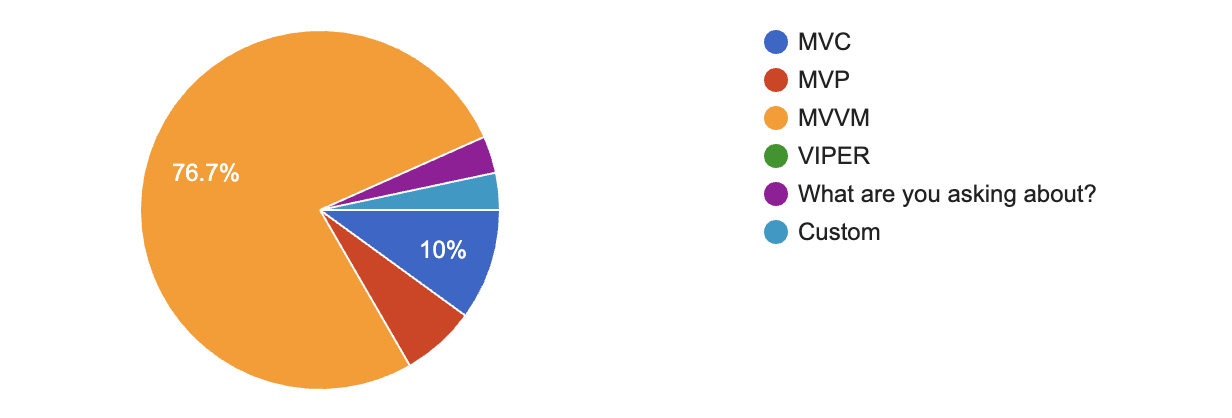
On the other hand, there is no fixed solution for that problem that Android demands from developers to meet its applications and there are a variety of options. So, the right way to architect Android apps still remains in discussion with clashing standpoints which are generally affected by technological hypes(19). The reviewed literature and the information gathered from industry also shows that the debate on choosing the right architectural pattern for an Android application remains quite controversial. Moreover, as a part of this study, a survey conducted amongst the professional Android developers shows that tendencies of the Android community are mainly focused on three main presentational patterns. The main purpose of this Android developer survey was to gather the most up to date data regarding the tendencies of the Android community as the tendencies and technological hypes change quite fast in the Android environment. Since the study topic is tightly bonded to the industry, it was also important to gather information from the Android developers who work in the industry, in addition to the academic literature review, in order to assemble the most up to date and most reliable information, regarding the study topic and the existing solutions to the problem that the study is trying to solve. The interpretation of the literature review and the developer survey is below.

A systematic review through the related academic resources shows that MVP, MVVM, and Clean Architecture are the most attributed design patterns and architectural patterns. MVP and MVVM are 2 different derivatives of the MV+X model. They are widely used in GUI-heavy software applications. Although ranked second in the results, MVVM is the recommended architectural pattern by Google as Google’s Android team released Architecture Components as a part of Android Jetpack. Google encourages Android developers to use Architecture Components and MVVM as their architectural pattern and it provides lots of guides for this aim (27). The Clean architectural pattern shows up to be also often considered by developers with regards to architecting Android applications. The remaining architectural patterns can be ignored as they are quite outdated. Another study that conducted a survey between Android professionals and a systematic review made amongst academic literature makes these architectural patterns known. A list of these patterns is shown in the figure below(19).



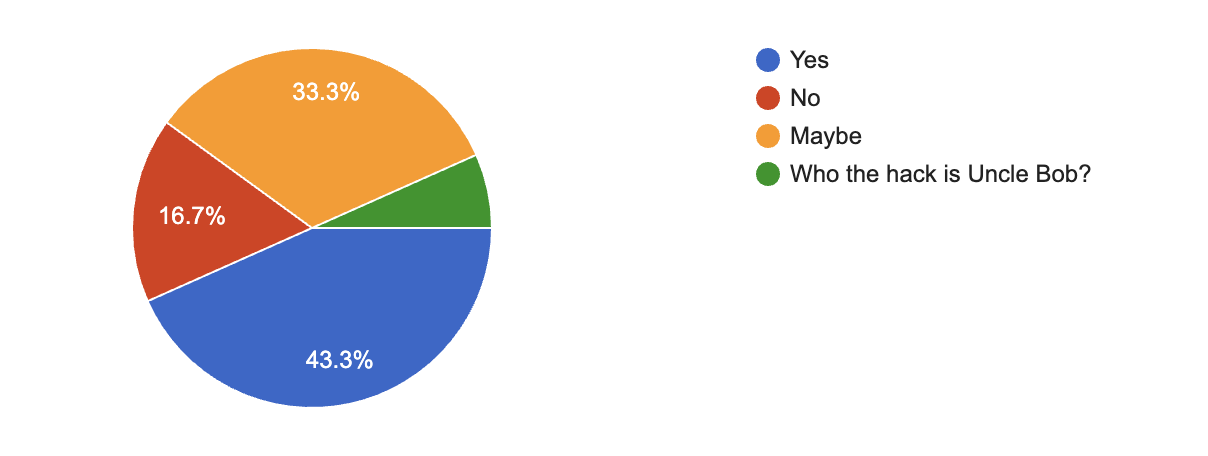
**Figure 4.** Well-known architectural and design patterns for Android (19)

When it comes to interpreting the results of the Android developer survey, it is seen that the outputs are mostly the same but with slight differences. According to the result of this Android developer survey conducted amongst more than a hundred Android developers, MVVM is the presentational design pattern that dominates the tendencies of Android developers. As an answer to the question "What presentational design pattern do you apply to your Android apps?", 76.7 percent of all participants responded that MVVM is their presentational design pattern of choice. MVVM is followed by MVC(10%) and MVP(2%). It seems that the recently introduced library called "Android Jetpack" which was designed by Google to help Android developers structure Android applications based on the MVVM design pattern has had a huge impact on the Android community lately. Consequently, MVVM is the top choice of Android developers according to the Android developer survey. And this outcome is different from the results identified by the academic literature review, where MVP was the most attributed one. This situation might be interpreted in such a way that the academic literature has been following the changes in the Android environment a bit slower.



**Figure 5.** Design pattern choice by Android developers according to the developer survey conducted by the study.

As a part of the Android developer survey, the question "Do you apply Uncle Bob's CLEAN Architecture to your Android applications?" was also asked. The answers show that only 43% of all participants apply Clean Architecture and its principles when developing Android applications. 33.3% of the participants responded that they might consider applying and 16.7% of the participants say that they do not apply Clean Architecture and its principles when developing Android applications.



**Figure 6.** Usage of Clean Architecture by Android developers according to the developer survey conducted by the study.

Both academic resources and information gathered from the industry indicate that, when it comes to building Android applications with high maintainability, the architectural choices are shaped around the same main design patterns. However, When the existing studies are examined, most of the studies cover the basic implementation of these design patterns and comparisons between these design patterns in terms of performance and maintainability. The systematic review indicates that the lack of detail and outdatedness in these studies is apparent

**2.6 Summary**

In this section, the insights of complexity issues in software development and the consequences of the complexity issues in software development were covered in general. Also, it was indicated why the challenges get even more complex when it comes to Android application development. To that aim structural information about the Android environment and Android applications were provided. Moreover, software architecture and maintainability were discussed as a solution to overcome the software complexity as a whole and also in the context of Android. In order to identify the possible solutions in the context of Android development, a systematic academic literature review was made and an Android developer survey was conducted. Results from the academic literature review and an Android developer survey were interpreted and shared under this section as well. To summarize, in this section, it is aimed to give information about software engineering and Android related concepts that should be known in order to better understand the problem that this study is trying to solve, and the information required for this purpose is shared as much as possible.

**3. Problem Description**

In this section, the problem that the study is trying to solve will be explained and the research questions will be defined. In addition, other studies previously conducted in the field covered by this study and the results of these studies will also be discussed.

**3.1. Problem Statement**

In the previous sections, the difficulties encountered while developing Android applications were mentioned under 3 main topics:

**Android platform-specific complexity:** Android applications need to fulfill some platform-specific requirements. Apps should work in accordance with Android OS and they have to use classes and structures offered by the Android Software Development Kit such as Activity, Fragment, Service, etc. A couple of important headlines from these platform-specific requirements were shared in the previous section. The flexibility of the application is directly affected by this situation and it limits the developers to use some certain software engineering techniques specific to Android development. It is clear that these components that are directly related to the Android OS should be considered as a different layer in the architecture of the Android application. These components should be separated from the other possible layers to decrease the level of complexity and increase maintainability.

**Business-specific complexity:** Since Android applications have started becoming more business-critical the level of their complexity has increased (2). This business-specific complexity along with the platform-specific complexity that we mentioned above makes developers’ jobs even harder as its influence on development is high. In this regard, it is not hard to notice that the classes which have the responsibility of the business logic should be independent of the other responsibilities. Business logic related classes and interfaces seem like a good candidate for becoming another layer when architecting Android applications.

**High update rate:** Due to changing business requirements, the addition of new features to the applications, and also the bug fixes, the Android applications have a high update rate and a quite active software development life cycle(3). As Robert C. Martin states in his book “Clean Architecture”, the primary goal of software architecture is to assist the life cycle of the software system. The right software architecture helps to create a software system that is easy to develop, maintain, understand, and deploy and it facilitates to do all these in a timely and cost-efficient manner(11). Hence, considering the active software development lifecycle of an Android application, it is not hard to see that the architectural choice is very important in terms of time and cost. This situation has a direct effect on the way Android applications are developed. Moreover, with the other two challenges mentioned above, the situation gets even more complicated and the importance of the architectural decisions for Android application development increases In order to decrease maintenance costs, shorten release times and improve developer efficiency, a well-architected Android application is fundamental. Only with a well-architected Android application, surviving the competitive Android market would be possible(19).

In the context of software engineering, maintainability means how well a software system is understandable, repairable, and extendable. Maintenance is one of the most important parts of the software development life cycle because the time spent on maintaining software systems requires more time and resources than the rest of the process. The relative expense for maintaining software and dealing with its development speaks to over 90% of its absolute expense (4). The importance of the maintainability for software systems is evident and this situation is no different for Android Applications. In addition to that, in the context of Android, when the three main challenges mentioned above are evaluated together, the importance of maintainability as a non-functional requirement becomes even more evident for Android application development, because the high level of maintainability is the way to overcome the challenges and complexities mentioned above while developing Android applications. In consequence, the question is how to achieve the goal of developing Android applications with high maintainability. From the software development point of view, the Android platform does not have any strict rules on the way the applications are developed, developing maintainable applications is not an obligation as previously stated in the background section. However, as it was already explained formerly, developing Android applications with high maintainability is a must in order to solve the mentioned difficulties above in a timely and cost-efficient manner, to facilitate the development processes for the Android developers, and to increase the quality of the Android applications. The methods to be followed and the technologies to be used in the development of Android applications for meeting these requirements have evolved in the course of time and the topic is still controversial among the Android community. Different solutions have been proposed and tried since the birth of the platform. However, the unchanged reality is that the most important criteria for building software systems with high maintainability is architecture selection. In other means, good architecture is what makes a software system maintainable and of course, this reality is not different for Android applications.

There are well-known design patterns amongst the Android community that targets to solve maintainability issues of Android. These well-known design patterns were already mentioned in the background section. As a reminder from the previous section, a couple of remarkable ones amongst these design patterns can be listed as Model-View-Controller(MVC), Model-View-ViewModel(MVVM), Model-View-Presenter(MVP).

In almost all of the examined academic studies and the reviewed work from the industry, the fact that these design patterns are presentational design patterns and they are not architectural patterns is completely ignored. In other means, the patterns derived from the MV-I concept, such as MVVM, MVP, MVC, are actually designed for how data is managed for display purposes. MV-I design patterns are intended to control the communication between the view layer and the data layer of GUI heavy applications. However, while developing small-sized Android applications, these design patterns can be effective up to a point. Nevertheless, while developing enterprise Android applications that have more sophisticated and complex business logic, it is obvious that these design patterns will be insufficient in scaling the application and solving the maintainability problems we have mentioned. Furthermore, this part will be covered in the upcoming sections in detail. Hence, the visibility of the need for higher-level architecture when developing complex Android applications is clear. It is wise to have deep knowledge for an Android developer or researcher regarding a higher-level architecture pattern, which can define a way of organizing complex Android applications. In this study, it is aimed to provide extensive knowledge of how the Clean Architecture, which is a candidate to be one of these high-level architectural patterns, can be used when developing Android applications by answering the following research questions below:

* How Clean Architecture can be adapted to Android application development using the latest Android development technologies and principles?
* What is the impact of adapting Clean Architecture on the Android application development in terms of software maintainability?

In order to answer the first question, insights, and experiences from real-world Android applications that have been developed following the principles of Clean Architecture will be shared. This Android application has been developed with the help of the latest Android technology stack including in addition to the Clean Architecture principles. In the light of this knowledge and experience, how Clean Architecture can be applied while developing Android applications and the roles of popular Android technologies in this process will be elaborated and implementation details will be shared.

To answer the second research question, the maintainability of the Android applications that developed based on Clean architecture was compared based on some metrics to other possible approaches. Also, the opinions of the experienced Android developers from the industry regarding the impact of Clean Architecture on Android application development in terms of maintainability and developer satisfaction will be shared.

**3. State-of-the-art Android Applications with High Maintainability**

**3.1 Literature Review**

**3.2 Existing Solutions**

**3.3 Android Developer Survey**

**3.4 SOLID Principles**

**3.5 CLEAN Principles**

**3.6 Dependency Injection**

**3.7 CLEAN Architecture**

**3.8 MVVM, MVP, etc**

**Factors of Maintainability SOLID**

**3.9 Libraries and Tools**

**4. Application Details**

**4.1 Overview**

**4.2 Functional Requirements**

**4.3 Qualitative Requirements**

**5. Implementation Details**

**6. Evaluation and Comparison**

**6.1 Readability**

**6.2 Extendability**

**6.3 Testability**

**6.4 Results**

**6.5 Thoughts and Comments From Industry**

**7. Conclusion**

**7.1 Thesis Summary**

**7.2 Future Work**