**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. Today, there are more than 2.5 billion active Android devices in the world (1) and smartphones and mobile applications became the people’s main way of interacting with technology. This situation has made the applications that work on these smartphones a very important part of daily life and business life from ordinary people to large companies. Thus, mobile application development has become an important topic in the IT industry and academy. This change also brought more challenges to mobile application development as applications get more and more complex. As the complexity of mobile applications increases and mobile applications become more business-critical, it is a must to apply software engineering processes to make mobile applications secure and high-quality (2). 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.

Android applications are distinguished from traditional web and desktop software with their complex features, unique characteristics, and structure specific to the Android platform. For the reason that a typical Android application consists of activities, fragments, services, broadcast receivers, network elements that communicate with the outside world, local databases, content providers, and many other platform-specific cases. The necessity of working together in harmony for all these Android elements, the challenges that arise from the Android operating system and Android SDK's nature, the unpredictable life cycles of some Android elements, and the limited resources of mobile devices make Android applications complex software structures that are hard to develop.

The high demand for Android applications and the high frequency of updates are other challenges when developing Android applications. These challenges make the development of high-quality and maintainable applications essential for Android application developers(19). Because users demand error-free, high-performance, easy-to-use, and low-energy applications. Developers, on the other hand, aim to develop maintainable, expandable, scalable and easily testable applications in the shortest time due to the rapidly changing and evolving user requests, and to be able to update the application as soon as possible to future user requests(21). Also, it gets harder to maintain the codebase as the codebase and the development team grows. Any time a new developer joins the team, the time required to onboard the new developer to the codebase is directly related to level of readability and maintainability of the codebase. Therefore, developing high-quality Android applications to meet all these expectations, overcome the mentioned challenges, and delivering the application rapidly is essential. Thus, processes such as updating Android applications, adding new features, fixing errors will be more time and cost-efficient in terms of software engineering principles and software quality standards.

Considering the complexity of the Android platform, the difficulties arising from the business requirements and the rapidly developing technology, a very important question is how to develop high-quality Android applications with the capacity to overcome all these difficulties and challenges. When the sources from the industry are examined, different sources answer this question in different ways as there is no certain way of answering such a question. In addition, when academic resources are examined, the inadequacy and outdatedness of these resources immediately become apparent.

Over the last decade, a couple of different ideas were in point to resolve these issues in the context of Android application development. Some of the remarkable ideas amongst these ideas can be counted as Model-View-Controller, Model-View-Presenter, Model-View-View-Model and VIPER. These are the well known design patterns in the industry for Android application development(19). The main purpose of these architectures is to overcome the issues and challenges we have mentioned above. Although these architectures work well for separating the business and presentation logic from the view, they are not enough when an Android app gets bigger and the codebase becomes more complex. When the codebase becomes huge, the presentation related classes become bloated and applying separation of concerns becomes very hard. Therefore, we see that these solutions are not sufficient to overcome the difficulties mentioned above, especially when it comes to the development of large and enterprise Android applications. As this information shows, it is clear that in the process of developing complex enterprise Android applications, a more advanced solution is needed to resolve the mentioned concerns. Bob Martin has described a set of rules and principles in his book “Clean Architecture” to improve the separation of concerns and increase the maintainability of the software systems (11). Clean Architecture, based on SOLID principles, is a high-level guideline for creating software systems with a layered architecture. The main purpose of the Clean Architecture is to make a software system more understandable and maintainable. The same set of rules can be applied to Android application development to resolve the previously mentioned issues.

This study aims to provide detailed information regarding the development of Android applications with Clean Architecture to overcome the maintainability and complexity issues for large and enterprise Android applications. It is also studied why Clean Architecture is the solution and what impact it has when solving the maintainability problems in Android application development, what are the pros and cons of applying Clean Architecture to the development of an Android application. In addition to the theoretical information regarding the application of Clean Architecture principles to Android, this study shares the best practices from the industry with elaborate examples. In addition, insights from the most popular Android libraries and how these libraries can be adapted to the Clean Architecture in Android are given. 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 challenges mentioned in section 1.1. The goal of this study is to provide complete, detailed, and up-to-date resources for how to develop high-quality and state-of-the-art Android applications with high maintainability to overcome these challenges by the guidance of Clean Architecture. Moreover, it also aims to provide detailed information about the tools, libraries, and techniques that can be used to achieve the goal.

To achieve this goal the best and most up to date practices applied when developing a real-life Android product in top companies from the industry will be identified, studied, and shared. The identified and studied practices will be applied to a sample Android application. Lastly, as a part of this study, the impact of Clean Architecture on the maintainability of the Android applications will be evaluated. For the purpose of this evaluation important maintainability factors such as code readability, testability, expandability, and separation of concerns and developer satisfaction will be measured. This measurement will be used to understand how the researched principles and technologies in this study are beneficial in Android application development to improve maintainability.

**1.2 Contributions**

The main contribution of this study is providing comprehensive information for the development of large scale Android applications with Clean Architecture and popular 3rd party Android application development libraries. With the help of the information that this study offers, developers and researchers will be able to have knowledge regarding the fundamental principles and technologies required to develop large-scale Android applications with high maintainability that can be considered state-of-the-art.

In view of the fact that similar studies in the academy 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 the impact of Clean Architecture from the maintainability point of view on Android application development will provide empirical data. As a part of the evaluation process, interviews will be conducted in order to hear views of experienced Android developers who work in the industry. Including the opinions of experienced Android developers regarding the impact of Clean Architecture on the maintainability level of the Android applications to this evaluation results, will make this data more valuable. Last in order but not of importance, a survey will be conducted to Android developers working in the industry to identify popular technologies, which are mentioned above in the main contribution, to be used when adapting clean architecture to the development of Android applications. The information to be collected through this survey will facilitate for the target audience of this study to be informed about the latest technologies used in the Android community.

**1.4 Thesis Outline**

The rest of this study is structured as follows:

**2. Background:** General information about the Android environment will be provided in this section. In addition, software maintainability issues in Android application development, and reasons for these issues will be covered in general. Moreover, well-known design patterns and architectural approaches to overcome the maintainability issues in Android application development will be introduced with their pros and cons. Lastly, the reason for the need for a higher level architectural approach when developing complex, enterprise Android applications will be explained in the summary of the section. In this section, previous studies similar to this study, and the results of these studies will be covered as well.

**3. Problem Statement:** The problem that this study targets to solve will be explained detailly in this section. Besides, an overview of the related studies and literature will be provided.

**4. Adapting Clean Architecture to Android Development:** Detailed theoretical information regarding Clean Architecture will be provided in this section. Likewise, the information regarding the adaptation of Clean architecture will be available in this section. Technologies and third-party libraries needed in order to adapt Clean Architecture to Android development were determined through an Android developer survey as a part of this study. The results of this survey and information concerning the determined technologies will be presented in this section as well.

**5. Implementation Details:** Use cases of the sample Android application which will be implemented throughout the study will be explained in this section. On top of that, comprehensive information about the implementation of the Clean Architecture principles with the help of the previously determined technologies in a sample Android application will be given in this section with code samples and explanations.

**6. Evaluation:** The maintainability factors and the methods used for evaluating these factors will be explained in this section. Also, the results of the evaluation will be shared along with thoughts and comments on the evaluated technology stack and principles, from the experienced Android developers who have been using these technologies and principles in real-life Android applications.

**7. Conclusion:** A condensed epitome of the study along with the thoughts and comments will be discussed in this section. Ultimately, future study opportunities related to the study topic of this thesis will be addressed.

**2.Background**

***“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 (11)***

***“Of course bad code can be cleaned up. But it’s very expensive.” - Robert C. Martin (16)***

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 "maintainability", due to the fact that this kind of 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 rather than being maintainable(17).

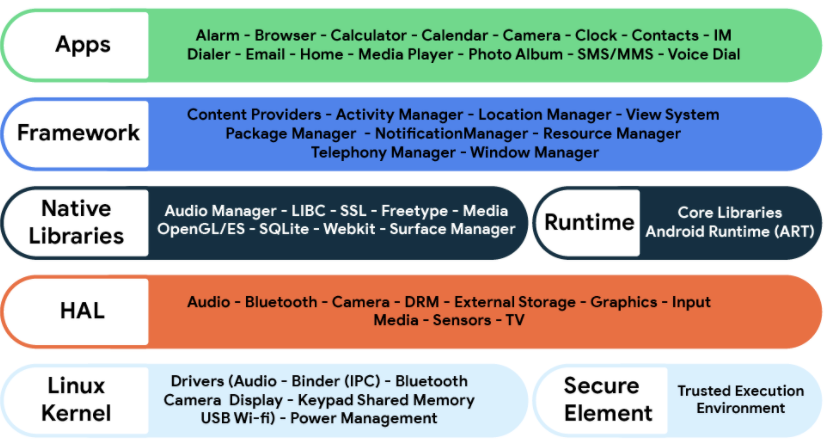
Fortunately, this situation has changed in the current day. With the birth and growth of the programming paradigms such as Object-Oriented Programming and also developments in the field of hardware, the priorities have altered. Even a low-quality mobile device is much stronger and smarter than the computers that we were using a couple of decades ago. Depending on these developments, software products have also become complicated and more functional(17). Today, in the view of the fact that the time has to be spent on maintenance of complex software products is comparatively longer than the rest of the software development lifecycle processes, maintainability became one of the most important aspects that should be taken into consideration when developing complex software products with teams. Reports indicate that the amount of effort spent on software maintenance is between 65% and 75% of the total amount of effort(18). Android application development is no exception for considering maintainability as an important aspect of the software development lifecycle given that the growing user demands and business needs making the mobile applications more and more complex.

This section starts with brief insights from the Android environment and Android application components in order to state the possible restrictions on the maintainability of Android applications that are caused by Android's nature. Moreover, the section reviews the software maintainability issues in general and discusses why maintainability is one of the most important aspects in software development and particularly in Android. Lastly, solutions to maintainability issues in Android and related studies are examined.

**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).

**2.1.4 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 communication between the device hardware and framework. The hardware includes important device features such as Bluetooth, camera, sensors.

**2.1.5 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):

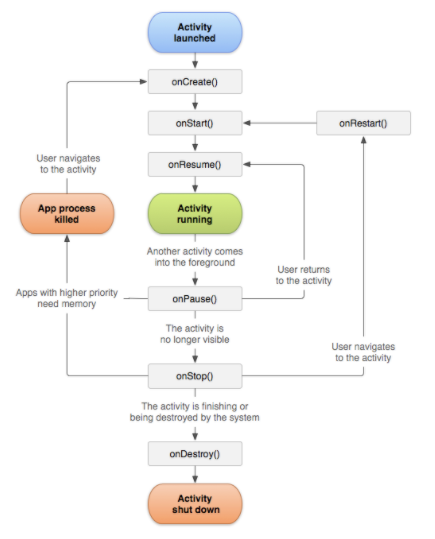
* Activities
* Services
* Broadcast receivers
* Content providers

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 the 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.



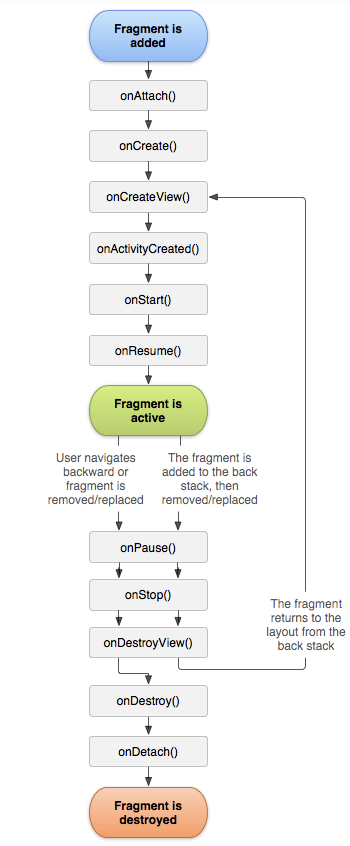
**Figure 2.** Android activity lifecycle (10)

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 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 Software Architecture and Maintainability**

Previously in this study, it is above mentioned that with the development in the areas of hardware and software, the priorities in the software development and the way the software products are developed have changed. 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 major failures.

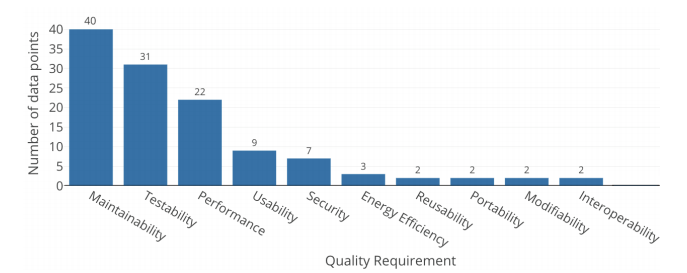
In his famous book "Clean Code", Robert C. Martin explains how a very popular company in the late 80s was wiped out from the business because of a poorly managed code organization. When the release cycles of their prominent product extended, due to the unorganized code base of their product, they were 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 and bad code was the reason for this company to go out of the business(16). This real-life example clearly indicates that software complexity is a fact and trying not to overcome that fact might be a very deadly decision.

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". 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 become 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 to 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.

**2.4 Existing Solutions for Maintainability of Android Applications SOLID**

**2.5 Literature Review**

**2.6 Summary**

**3. Problem Statement**

In the introduction, 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. The flexibility of the application is directly affected by this situation and it limits the developers to use some certain software engineering techniques. It is clear that these classes that are directly related to the Android OS should be considered as a different layer and they should be separated from the other possible layers to decrease the level of the complexity and increase the 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.

**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 life cycle 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 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. Considering that 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 3 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. However, as we already explained the reasons why, developing Android applications with high maintainability is a must in order to solve the mentioned difficulties above in a timely and cost-efficient manner, facilitate the development processes for the Android developers, and increase the quality of the Android applications. The methods to be followed and the technologies to be used in 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. 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), and VIPER as we already referred to in the previous section. When the existing studies are examined, it can be easily seen that these are the most indicated design patterns for Android. Most of the studies cover the implementation of the basics of these design patterns and comparisons of these design patterns in terms of performance and maintainability. After a systematic review, the lack of detail and outdatedness in these studies become apparent. In addition, in almost all of these studies, the fact that these design patterns are presentational design patterns and they are not architectural patterns is completely ignored. 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 and it is wise to have deep knowledge for an Android developer or researcher regarding a higher-level architecture pattern. 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, a small Android application that will cover the principles of Clean Architecture was developed with the help of the latest Android technology stack including RxJava 3, Dagger 2, Retrofit, Gradle, etc. The main purpose of developing this sample Android application is to provide the implementation details of an Android application that is developed based on Clean architecture. 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**