UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ALAN SALAZAR WINK

Android Platform Analysis from the Software Engineering perspective

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engeneering

Advisor: Prof. Dr. Érika Cota

Porto Alegre
June 2016

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb
Coordenador do Curso de Engenharia de Computação: Prof. Raul Fernando Weber
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“An american pupil who is good in French
can say "Please pass me the salt" gets an A,
while a French child simply gets the salt!”

— B. F. Skinner

ACKNOWLEDGEMENTS

I would like to thank my family, for making this dream come true.

I also would like to thank my advisor, professor Érika Cota, who accepted my idea and helped me to make this project become real.

And also, thank my colleagues, that helped me during these years to keep moving forward and always improving professionally, as well as personally.

ABSTRACT

Applications for mobile platforms become increasingly complex with increasing market of mobile devices. To avoid an increase in development costs, it is necessary to use Software Engineering techniques. However, mobile devices have a set of non-functional requirements different than desktop platforms, such as power consumption, data usage, etc. There is also the need to follow restrictions imposed by the platform for developing applications, requiring techniques and models of Software Engineering to be adapted to use in this platform.

The quality of applications developed in a platform depend directly of the own platform quality, and from the resources that are available to the developer. However, there are techniques that can be used by the developer to improve the quality of their applications, using resources that were not originally available in the platform.

In this study, we analyzed the Android platform and the components offered to the developer and how these impact the quality of the final application from the point of view of Software Engineering. We also verified through the development of a mobile app to how the developer can apply Software Engineering techniques to enhance the developed application. We also analyze alternative components that seek to improve the features offered by the platform and consequently its quality.

Keywords: Android. Software Engineering. Embedded Devices. Mobile Devices.

Análise da plataforma Android da perspectiva de Engenharia de Software

RESUMO

Os aplicativos para plataformas móveis tornam-se cada vez mais complexos com o aumento do mercado de dispositivos *mobile*. Para que esse crescimento não implique no aumento nos custos de desenvolvimento, faz-se necessário o uso de técnicas de Engenharia de Software. Entretanto, dispositivos móveis possuem um conjunto de requisitos não-funcionais diferentes dos requisitos de plataformas *desktop*, como consumo de bateria, consumo de dados, etc. Existe também a necessidade de se seguir restrições impostas pela plataforma para o desenvolvimento de aplicativos, fazendo com que técnicas e modelos da Engenharia de Software sejam adaptados para uso dessa plataforma.
A qualidade do desenvolvimento de aplicativos para uma plataforma depende diretamente da qualidade da própria plataforma, e dos recursos que são disponibilizados para o desenvolvedor. Entretanto, existem técnicas que podem ser utilizadas pelo desenvolvedor de modo a melhorar a qualidade de suas aplicações, utilizando-se de recursos que originalmente não estariam disponíveis na plataforma.
Nesse trabalho, analisamos a plataforma Android e os componentes oferecidos ao desenvolvedor e como esses impactam na qualidade da aplicação final do ponto de vista da Engenharia de Software. Verificamos também através do desenvolvimento de um aplicativo como o desenvolvedor *mobile* pode aplicar técnicas de Engenharia de Software para melhorar sua aplicação. Também analisamos componentes alternativos que buscam melhorar os recursos oferecidos pela plataforma e consequentemente sua qualidade.

Palavras-chave: Android, Engenharia de Software, Sistemas Embaracados, Sistemas Móveis.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
MVC	Model View Controller
MVP	Model View Presenter
SDK	Standard Development Kit
SQL	Structured Query Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language

LIST OF FIGURES

Figure 2.1 Android Activity lifecycle. 14
Figure 2.2 Content provider example.. 15
Figure 2.3 App composition with fragments. 17
Figure 2.4 XML resources example.. 18

Figure 3.1 The MVC model. 20
Figure 3.2 The MVP model. 20
Figure 3.3 McCall's Quality Factors. 21
Figure 3.4 ISO 25010 quality factors. 23
Figure 3.5 Franke et. al. mobile quality model. 24

Figure 4.1 The Android model, when mapped to the roles in MVC and MVP models. 27
Figure 4.2 The Context class and its inheritances. 28
Figure 4.3 AlertDialog example.. 30
Figure 4.4 An Intent usage example in an email application. 32
Figure 4.5 Activity and Fragment lifecycles. 34

Figure 5.1 Screenshots of PinMaps.. 37
Figure 5.2 Version 1 - Original model for PinMaps. 38
Figure 5.3 Version 2 - Model and controller coupling. 38
Figure 5.4 Version 3 - View coupling, reducing all views to one class. 39
Figure 5.5 Version 4 - View and Controller coupling. 39
Figure 5.6 Version 5 - Full coupling. Removal of all unecessary methods by replacing the method call with the actual code (inline expansion). 39

Figure 6.1 Screenshots of Topeka on API version 19 (left) and API version 23 (right). 43
Figure 6.2 EventBus diagram between one publisher and two subscribers. 44
Figure 6.3 Realm database example.. 45

LIST OF TABLES

Table 3.1 McCall's Product Revision factors and criteria 22
Table 3.2 Quality factors selected for this analysis 25

Table 4.1 Selected methods from the Context class. 29

Table 5.1 Extracted metrics for each version of the app. 40

CONTENTS

1 INTRODUCTION **11**
2 ANDROID PLATFORM **13**
2.1 Basic application components **13**
2.1.1 Android Manifest 13
2.1.2 Activity 14
2.1.3 Services 15
2.1.4 Content Providers 15
2.1.5 Broadcast Receivers 16
2.2 Other components **16**
2.2.1 Fragments 16
2.2.2 Intents 17
2.2.3 XML Resources 18
3 APLICATION QUALITY **19**
3.1 Architectural Pattern **19**
3.1.1 MVC Model 19
3.1.2 MVP Model 20
3.2 Quality Factors **21**
3.2.1 McCall's Quality Factors 21
3.2.2 ISO 25010 22
3.2.3 Franke, Kowalewski and Weise Model 23
3.2.4 Developer perspective 24
4 ANDROID PLATFORM ANALYSIS **27**
4.1 Components organization **27**
4.2 Context class **28**
4.3 Communication with intents **30**
4.4 Content providers **32**
4.5 Fragments **33**
4.6 Custom Views **35**
4.7 Conclusion **35**
5 INTERNAL QUALITY ANALYSIS FOR A CASE STUDY APP **37**
5.1 App description **37**
5.2 Developed versions **37**
5.3 Extracted metrics **40**
5.4 Results **40**
6 QUALITY ELEMENTS OUTSIDE THE ANDROID PLATFORM **42**
6.1 Backwards compatibility support **42**
6.2 Communication alternatives **43**
6.3 Storage elements **44**
6.4 Testing elements **45**
7 CONCLUSION **47**
REFERENCES **48**

1 INTRODUCTION

The smartphone sales increase is changing the society relationship with tecnology. In less than 8 years Google Play Store (the app store for Android devices) increased from its initial 50 apps (TAKAHASHI, 2008) to more than 2 million apps (APPBRAIN, 2016). Today, more than 1.4 billion Android devices are active in the world (VINCENT, 2015).

This market increase changed the way users interact with technology. Mobile devices are becoming the main way people interact, in special accessing internet. In 2015, Amit Singhal, Google's Research Chief, announced that for the first time ever the amount of searches using mobile devices is greater than desktop devices (GRODEN, 2015).

This transformation on the consumer behavior makes mobile apps more important than ever. This made the apps more complex and with more use cases. After a research with mobile developers, Wasserman (2010), concluded that in that time, apps were small (some thousands lines of source code), having only one or two developers responsible for the app conception, design and implementation. However, the author himself warned about a problem that would happen for medium and large size apps:

> "... as mobile applications become more complex, moving beyond inexpensive recreational applications to more business critical uses, it will be essential to apply software engineering processes to assure the development of secure, high-quality mobile applications." (WASSERMAN, 2010)

The main difference between mobile development, compared to desktop platforms, is the need to meet some specific requirements from mobile devices, for example, integration with system apps, sensor reading, and especially non-functional requirements, as power consumption, network usage, adaptability to different screen sizes, etc. Another major difference is related to the restrictions imposed by the platform, since mobile apps need to follow the architecture imposed by the executor, causing a lower programming abstraction level when compared to desktop applications. Since the platform must be available to many devices, including low-end devices[1], it must require the minimum hardware possible. Therefore, the platform must impose some components that should be used by the developer in order to make the application work and be optimized, like Activities and Services, that will be discussed later.

The flexibility of the application is affected by the platform limitations. On these platforms, we cannot implement some Software Engineering techniques without an adaptation, or sometimes it is impossible due the limitations. Therefore, the platform can limit

[1] Inexpensive devices, usually with higher limitations on memory or processor.

the quality of the software developed by the resources it offers. In later chapters, we analyze the Android platform and the limitations it creates to the developer.

The raising of programming complexity level and the need to meet the specific requirements of mobile platforms make us think about software architectures and techniques that can be used without reducing the application's performance. With this study, we will analyze the Android platform as well as different techniques used by mobile developers to develop better applications from a Software Engineering perspective, increasing factors such as maintainability and flexibility, that will be discussed in later chapters.

The remaining of this work is divided as follows: Chapter 2 presents the main components of the Android platform and how they are used in an Android application. Chapter 3 discusses software quality, especially architectural models and quality factors that will be used in the next chapters. Chapter 4 discusses the Android platform characteristics and how it impacts developers and their applications. In Chapter 5 there is an example app developed in order to analyze the impact of the platform in Software Engineering good practices. In Chapter 6 there is a discussion of components outside the Android platform that help to increase the overall quality of the application. We finalize this study in Chapter 7 with some final thoughts and future work.

2 ANDROID PLATFORM

The Android platform, currently found in 76% of the smartphone market (AHONEN, 2015), is a multiuser Linux system, where each application is a different user to the system and has its own process with its own virtual machine to run its code. This system organization enhances security, since each application has access only to the resources that it is allowed (APPLICATION. . . , 2016).

2.1 Basic application components

The Android system architecture provides four basic components that enable the application to interact with the system. An Android application must implement at least one of these components to make the application executable by the system. All components implemented must be described in the application manifest, an XML file containing all the records of components and permissions that the application needs from the system. The Android basic application components are Activities, Services, Content Providers and Broadcast Receivers.

2.1.1 Android Manifest

The Android manifest is a XML file (*AndroidManifest.xml*) required in order to build an Android application. The manifest presents all the information required about the app to the Android system. Among other information, the manifest is responsible for (APP. . . , 2016):

- Defining the Java package name, which serves as a unique identifier for the application.
- Describing all the components of the application is composed of (Activities, Services, Content Providers and Broadcast Receivers). It also describes the intents the app is capable of handling.
- Listing all the permissions the app requires in order to work correctly[1].These permissions allow the app to access protected parts of the Android API, such as sensor

[1]Since Android 6.0 (API 23), the app also needs to request the permission at runtime. However, the permissions still need to be described in the manifest.

reading, network access, location reading, etc.

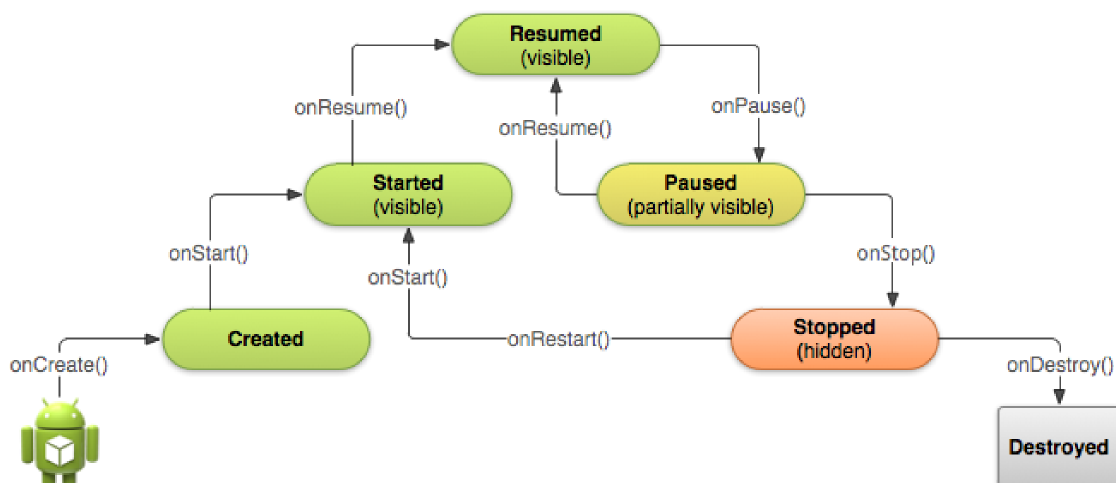
- Defining the minimum version of the SDK required. This protects older Android versions to install apps that have API calls that are not available in those versions.

The Android manifest can also be used by external applications. For example, Google Play Store reads the manifest in order to prevent users from downloading applications incompatible with their devices and to show the permissions the app requires (FILTERS. . . , 2016).

2.1.2 Activity

An Activity[2] represents a complete app screen. Usually, this component is responsible for loading the elements that will be part of the screen such as text boxes, buttons, and others. Activities have a life cycle related to the system, with methods that signal changes in application state, as showing in Figure 2.1. The activities are also responsible for receiving data coming from other system applications, such as a file that was chosen in another application, or a picture taken with the camera app that was sent to the app.

Figure 2.1: Android Activity lifecycle.



Available at: <http://developer.android.com/training/basics/activity-lifecycle/starting.html>

[2]Full documentation available at <https://developer.android.com/reference/android/app/Activity.html>

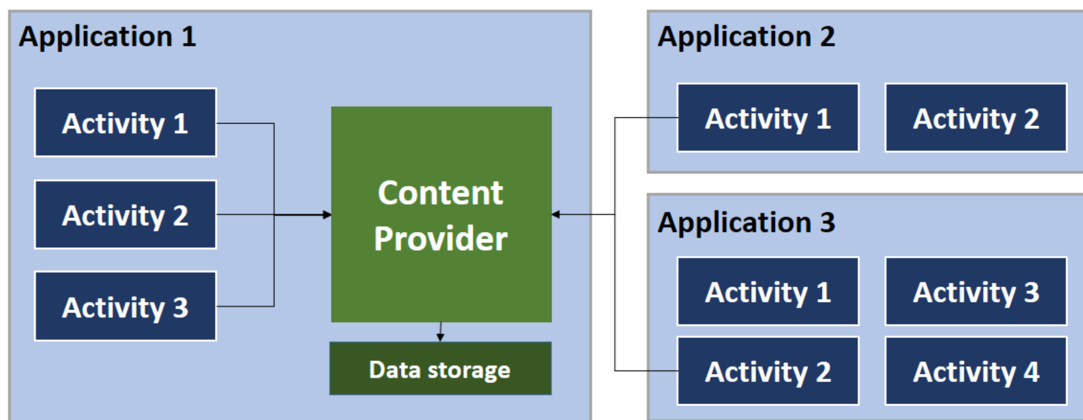
2.1.3 Services

Services[3] are components running in background that are used for long term operations. Services usually do not implement a user interface[4] and are responsible for operations that cannot be interrupted by the activity lifecycle (such as large downloads or media playback).

2.1.4 Content Providers

Content providers[5] are the components that manage the data shared with other apps, with options to define permissions to read and write, as depicted in Figure 2.2. They can also be used for managing private data within the application. Access to data in a content provider is made by an Uniform Resource Identifier(URI) that locates the data to be read, written or edited.

Figure 2.2: Content provider example.



Source: Author

[3]Full documentation available at <https://developer.android.com/reference/android/app/Service.html>

[4]A service can display information to the user, such as the progress of downloads or buttons to control media that the service is playing, usually as a notification in the notification drawer.

[5]Full documentation available at <https://developer.android.com/guide/topics/providers/content-providers.html>

2.1.5 Broadcast Receivers

Broadcast Receivers[6] are components that respond to messages indicating system events. These messages indicate global changes, such as when the system is charging, or when the system is connected to a wireless network. The messages that the application wants to receive are registered in the application manifest, and the receptors are triggered as soon as the registered event occurs.

2.2 Other components

Besides the basic components, the Android Platform offers many other components that can be used by the mobile developer. Some important components to our analysis will be discussed in this section.

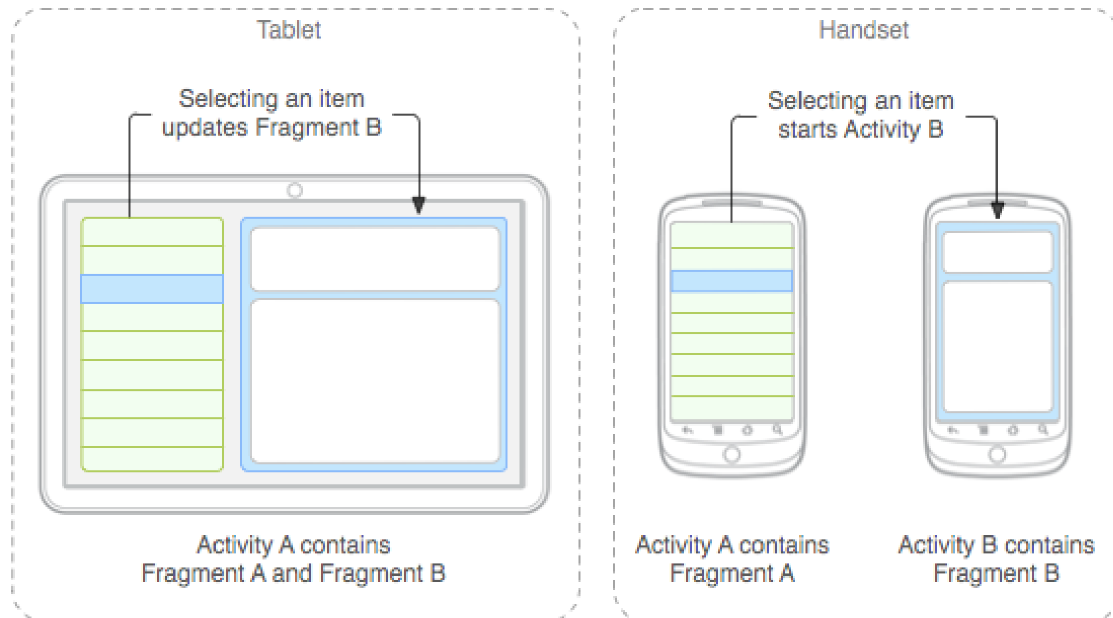
2.2.1 Fragments

Fragment[7] is a component introduced in Android 3.0 (API 11) and represents a portion of the screen. Unlike Activities, Fragments do not need to represent an application full-screen, allowing a it to be composed of different fragments, depending on the size of the device screen. This component is especially useful to develop for larger screens like tablets, without harming the layout for smaller devices. Devices with larger screen can display fragments simultaneously, while devices with smaller screens use the same fragments in different screens, as depicted in Figure 2.3.

[6] Full documentation available at <https://developer.android.com/reference/android/content/BroadcastReceiver.html>

[7] Full documentation available at <https://developer.android.com/reference/android/app/Fragment.html>

Figure 2.3: App composition with fragments.



Available at: <http://developer.android.com/guide/components/fragments.html>

2.2.2 Intents

Intents[8] are the main component responsible for communication between other Android components, allowing the exchange of messages between different system components. There are two types of intents: explicit and implicit Intents. Explicit Intents have a specific receiver component, and are used to perform communication within the app. Those intents are specially used to start new Activities and Services in an app. Implicit Intents contain the action the sender wants to be performed, giving to the Android system the decision about which application and component will be chose to perform the requested action[9]. Implicit Intents help the Android system to create an integrated experience to the user, sharing content between applications easily.

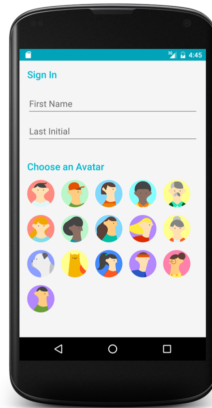
[8]Full documentation available at <https://developer.android.com/reference/android/content/Intent.html>

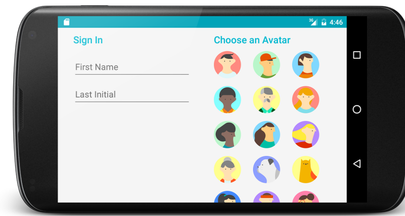
[9]Usually, when there is more than one application that can handle an Intent, the system prompts the user about which app must receive the Intent. For example, if a camera application shares a picture using an Implicit Intent, it can be send to a photo editor app to be edited, an email app as an attachment to a new message or to any other application that can handle this type of data.

2.2.3 XML Resources

The Android platform provides a flexible way to store application resources using XML files. These files externalize application resources like user interface layouts, color values, strings, animations, application values, etc. It is possible to provide alternative resources, depending on device variables such as screen size, system language, API version, etc. The alternative files are defined with the same name of the original file, but in a different folder with a qualifier name. The Android system detects the device configuration and selects the correct resource version to use. In Figure 2.4, we can see the application layout changes according to the device orientation.

Figure 2.4: XML resources example.





layout/fragment_sign_in.xml **layout-land/**fragment_sign_in.xml

Files in the folder *layout-land/* are selected when the device is in landscape mode. Source: Author

3 APLICATION QUALITY

Pressman (2005) defines Software Quality as "An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.". In this chapter, we will discuss some of these processes and techniques used to improve software and how they are related to mobile applications.

3.1 Architectural Pattern

One way to increase software quality is to define a well-based architecture, increasing application modularity and maintainability. The Microsoft Patterns & Practices Team defines it as a style that promotes reuse:

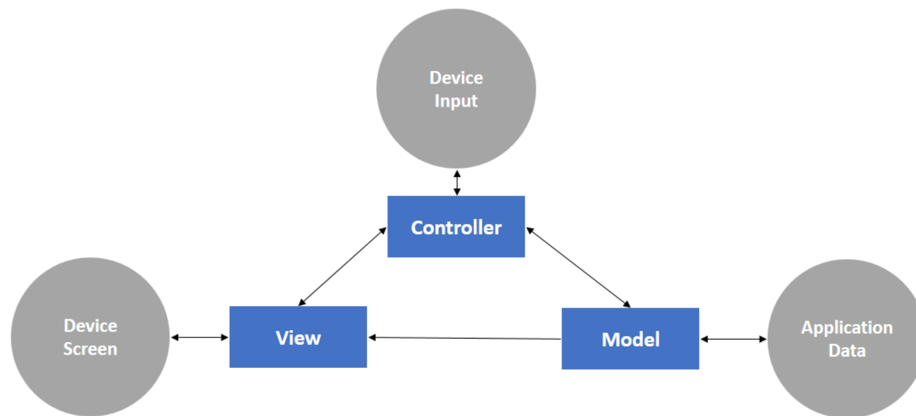
> “An architectural style, sometimes called an architectural pattern, is a set of principles—a coarse grained pattern that provides an abstract framework for a family of systems. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems.” (TEAM, 2009)

In this section, we will discuss some important architectural patterns widely used in software development.

3.1.1 MVC Model

Presented in 1978, the the MVC model is the oldest architectural model created (SOKOLOVA et al., 2014). This model is designed to separate business logic from presentation logic. It consists of three components: Model, View and Controller. The Model represents the application data. The View is usually a visual component, such as a button or a text field. The Controller handles input and communicates any changes to the View and Model. The Model also communicates with the View to update any modified data, without requiring to call the Controller. A visual representation for a mobile application is presented in Figure 3.1.

Figure 3.1: The MVC model.

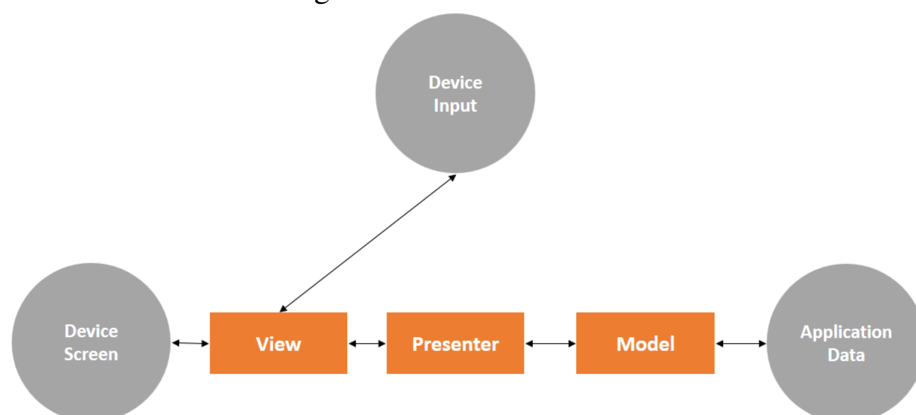


Source: Author

3.1.2 MVP Model

Introduced in 1996, the MVP model was proposed as an adaptation of the MVC model for event-driven system(SOKOLOVA et al., 2014). In this model, there are three components: Model, View and Presenter. The Model represents the application data, while the View represents a full screen. The Presenter is responsible by presentation logic, converting data from the model in the data to be shown by the View. A visual representation is presented in Figure 3.2.

Figure 3.2: The MVP model.



Source: Author

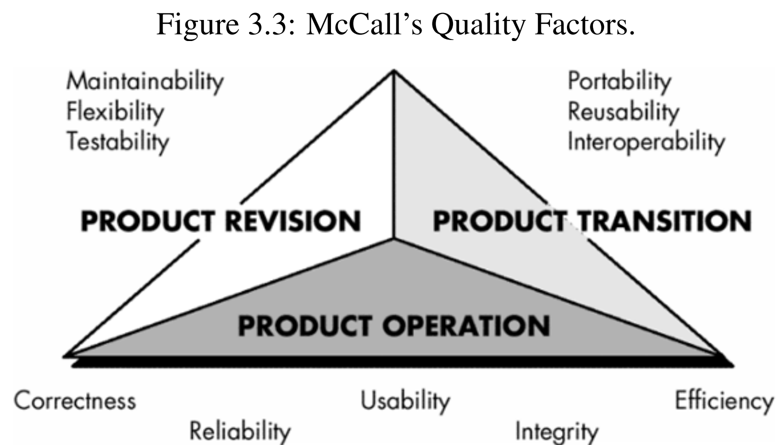
3.2 Quality Factors

One way to analyze the software quality is to understand the factors that influence the quality. In this session, we will discuss some quality factors to build a group of factors focused on mobile applications.

3.2.1 McCall's Quality Factors

McCall, Richards and Walters (1977) proposed a categorization of software quality factors focusing in three different aspects of software as a product, related to three aspects of the software development: its operational characteristics, the ability to be changed and its adaptability to new environments, as depicted in Figure 3.3.

Figure 3.3: McCall's Quality Factors.



Adapted from (PRESSMAN, 2005)

Marinho and Resende (2012) adapted McCall's quality factors with the criteria required to achive each factor. The result is shown in Table 3.1.

Table 3.1: McCall's Product Revision factors and criteria

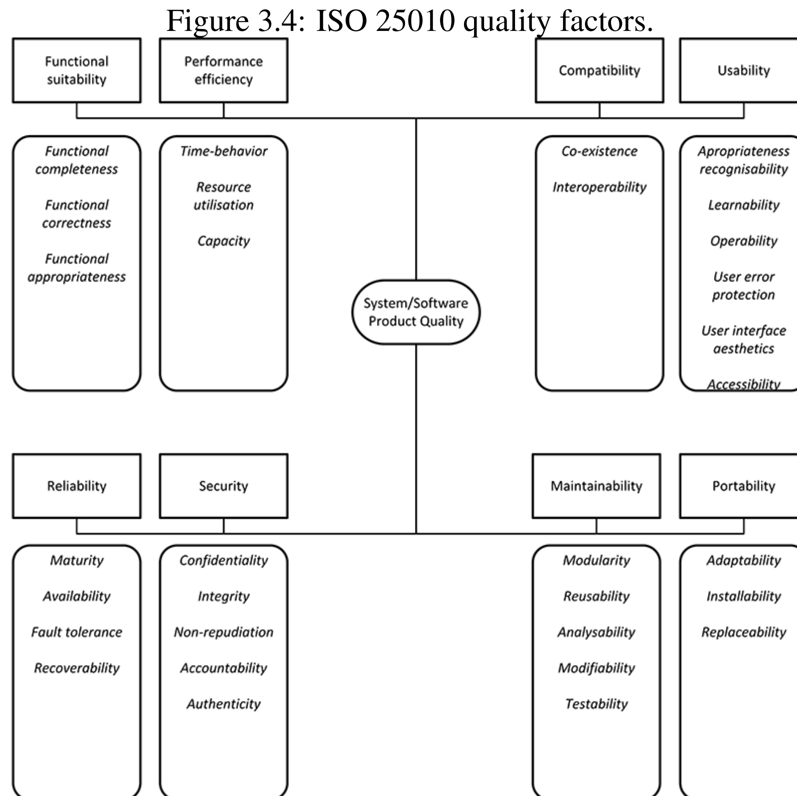
Product Activity	*Quality Factor*	*Quality Criteria*
Product Operation	Correctness	Traceability, Consistency, Completeness
	Reliability	Error Tolerance, Consistency, Accuracy, Simplicity
	Effciency	Execution Effciency, Storage Effciency
	Integrity	Access Control, Access Audit
Product Revision	Maintainability	Consistency, Simplicity, Conciseness, Modularity, Self-Descriptiveness
	Flexibility	Modularity, Generality, Expandability, Self-Descriptiveness
	Testability	Simplicity, Modularity, Instrumentation, Self-Descriptiveness
Product Transition	Portability	Modularity, Self-Descriptiveness, Machine Independence, Software System, Independence
	Reusability	Generality, Modularity, Software System, Independence, Machine Independence, Self-Descriptiveness
	Interoperability	Modularity, Communications Commonality, Data Commonality

Adapted from (MARINHO; RESENDE, 2012)

3.2.2 ISO 25010

The ISO 9126 (ISO/IEC, 2001), released in 2001, is a standard developed to identify key quality factors in computer software. As depicted in Figure 3.4, the standard has six key concepts: Functionality, Reliability, Usability, Efficiency and Maintainability. Most of these concepts represent quality from the user perspective. Later in 2011, the ISO 25010 (ISO/IEC, 2011) was released to extend quality factors to a broader system perspective.

Figure 3.4: ISO 25010 quality factors.



Adapted from (MARINHO; RESENDE, 2012)

3.2.3 Franke, Kowalewski and Weise Model

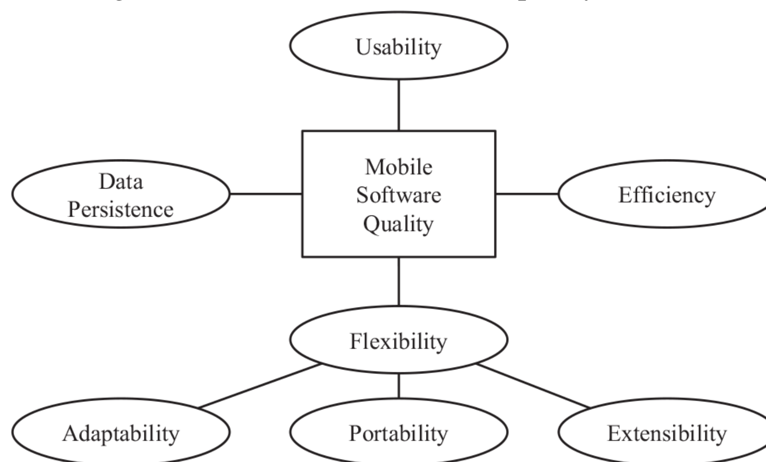
With the increase of mobile devices, quality models were created specifically to the mobile environment. The quality model presented by Franke, Kowalewski and Weise (2012) is depicted in Figure 3.5 and it has its factors focused on the mobile environment and its requirements as the adaptability to different conditions or the distribution by app stores. The quality factors defined in this model are (FRANKE; KOWALEWSKI; WEISE, 2012):

- *Flexibility:* The ease with which a system can be modified to be used in environments for which it was not originally designed. The app stores make the distribution to different devices very easily. Therefore, the developer must produce a software that works in different screen sizes, processors, sensors, etc.
- *Extensibility:* The ease with which a system can be extended. This includes functional extension (adding new functions to an application) as well as non-functional extensions, like changing the app design of underlying implementations.
- *Adaptability:* The ability to adapt to changes in the environment. For example, a

device that changes from a Wi-Fi network to a 3G network must change its network in a way the user does not notice.

- *Portability:* The ability to run the application on platforms other than it was originally written for. This is extremely important not only for different operational systems, but for different versions of the same operational system, as each version offers different resources to the developer.
- *Usability:* The capability of the software to be understood, learned, used and be attractive to the user. It can be related with the way the app interacts with the system, creating a uniform overall experience of the system.
- *Efficiency:* Mobile devices must use system resources (CPU, RAM, power consumption, ...) in the most efficient way, since those resources are limited in this environment.
- *Data persistence:* The usage of an application in a mobile environment is often a single app interaction. When an app is not visible, the system pauses it, and its instance can be destroyed to free some memory. When a call arrives, an alarm rings or the user simply changes the visible app, the current app is paused. Therefore, the app must persist its data to avoid data losses and create a better user experience.

Figure 3.5: Franke et. al. mobile quality model.



Source: (FRANKE; KOWALEWSKI; WEISE, 2012)

3.2.4 Developer perspective

Analyzing the quality factors presented in this chapter, we could observe that, although some factors are related to the development phase (such as Maintainability and

Reusability), most of the factors are related to the final product (like Efficiency, Security and Compatibility). In this study, we want to analyze the Android platform from a developer perspective. Therefore, we will focus on factors that are related to the software construction and maintenance. In the next chapters, we will see how the factors related to software design and construction are influenced by the need of an efficient and flexible platform.

To simplify our analysis, we will use eight quality factors divided in two groups: the first group has the factors related to the software project itself, the ones more important to the software construction and maintenance. The second group contains the factors related to the final product, and its execution on the final user device.

Table 3.2: Quality factors selected for this analysis

Project factors	*Product factors*
Flexibility	Performance
Reusability	Efficiency
Maintainability	Usability
Testability	Interoperability

Source: Author.

We define the project factors as:

- *Flexibility*: The ease with which a software can be modified in response to new requirements, as a feature addition or an architectural change.
- *Reusability*: The ability to use a pre-developed or existing software modules in new solutions.
- *Maintainability*: The ease with which a software can be maintained. In other words, the ease with which we can change the code to fix errors or to prevent them, instead of rewriting it.
- *Testability*: The ease with which a software can be tested.

We define product factors as:

- *Performance*: The relation between time, responsiveness and stability of a software.
- *Efficiency*: The way the software uses system resources (storage, execution, power) in order to accomplish a task.
- *Usability*: How easy it is for an user to interact with the software, and how it is integrated with the rest of the system, from a user perspective.

- *Interoperability*: The ability of a software to work on an environment different from the one it was designed to work.

In the next chapter, we will analyze the components the Android platform provides to the developer and how they influence the factors cited above.

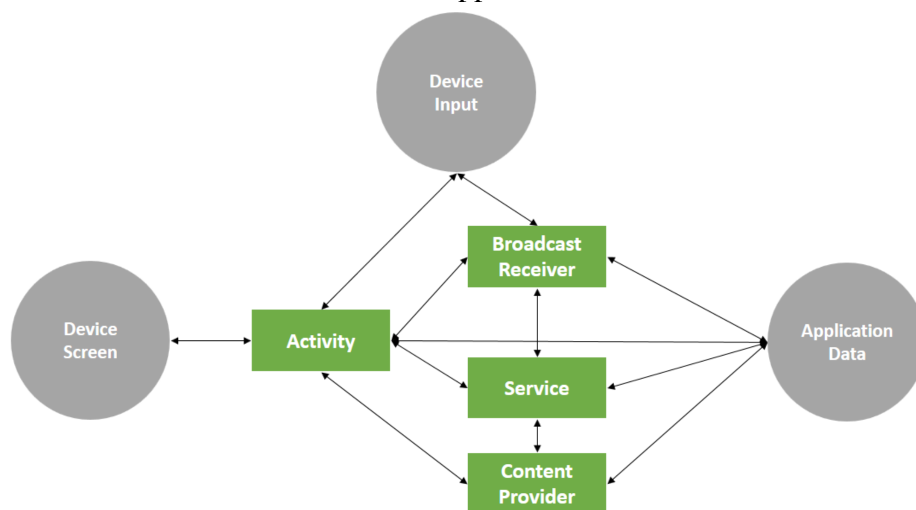
4 ANDROID PLATFORM ANALYSIS

The Android platform is the base for any application developed for it. Therefore, the way the platform is built influences the way developers will implement their apps. In this chapter, we will analyze the main structures of the Android platform, applying the quality criteria discussed in the last chapter.

4.1 Components organization

In Section 3.1, we presented some architectural models that are widely used by developers in other platforms. However, in the Android platform, there is no direct relation between the four basic components and a known architectural model, due the complexity of each module. For example, the Activity is responsible for the application screen, being a good candidate for the View entity in the MVC model. However, it is also responsible for receiving communication from other apps (using Intents) and its reference must also be passed to create databases. This situation is shown in Figure 4.1, where the Android components are mapped in the same way the entities of MVC and MVP are mapped. Therefore, the Android basic components cannot be mapped directly to an architectural model. In Chapter 5, we will adapt the MVC model using custom views in order to achieve flexibility and maintainability.

Figure 4.1: The Android model, when mapped to the roles in MVC and MVP models.

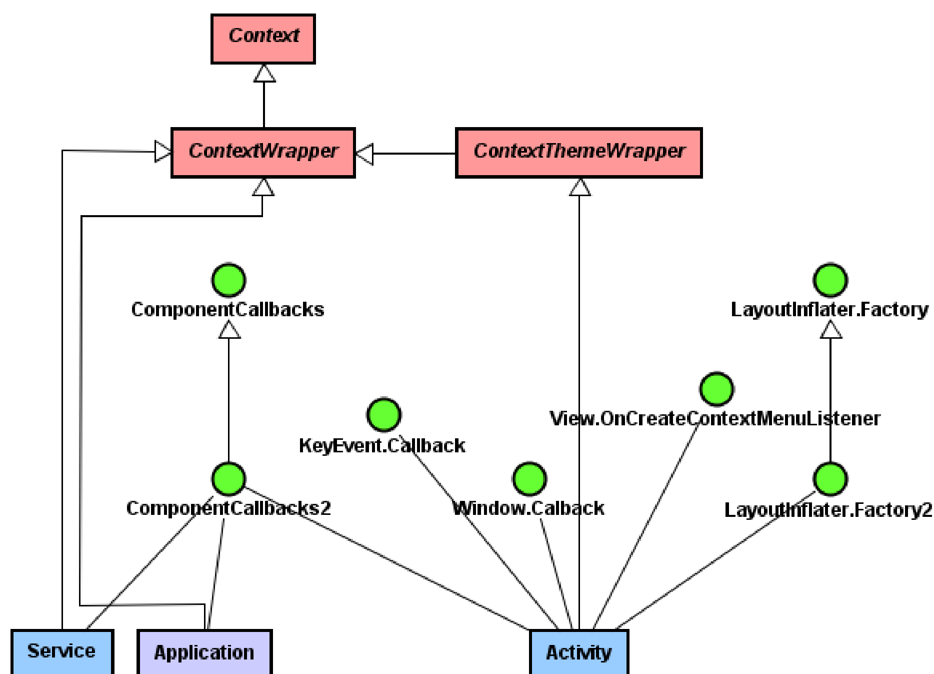


Source: Author

4.2 Context class

Activities and Services inherit from an abstract class called Context, as described in Figure 4.2. The Android documentation defines Context as: "Interface to global information about an application environment. This is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc."

Figure 4.2: The Context class and its inheritances.



Source: Author

Analyzing the methods in the Context class, we can see it has many methods that are used in a broad variety of use cases, from getting the list of databases of an application until getting the color from a XML file, as we can see in Table 4.1. This structure goes against the principles of modularity, since the cohesion of the class is extremely low. It also leads to a coupling problem: since the Context class has methods for many use cases, almost all the modules of the application require a reference to a Context, increasing the overall coupling, reducing the maintainability and the flexibility of the app.

Another problem found in the Context class is its division in different incompatible context types. The class Application (ANDROID. . . , 2016) is responsible for holding the global state of the application, tied to the whole process lifecycle, and not only to

Table 4.1: Selected methods from the Context class.

Method	*Description*
abstract String[] databaseList()	Returns an array of strings naming the private databases associated with this Context's application package.
abstract void enforceCallingPermission(String permission, String message)	If the calling process of an IPC you are handling has not been granted a particular permission, throw a SecurityException.
final int getColor(int id)	Returns a color associated with a particular resource ID and styled for the current theme.
abstract String getPackageName()	Return the name of this application's package.
abstract Object getSystemService(String name) abstract void startActivities(Intent[] intents, Bundle options)	Return the handle to a system-level service by name. Launch multiple new activities.

Source: (ANDROID. . . , 2016)

the component lifecycle, as the Activity and Service contexts. In the ContextWrapper[1] documentation for the *getApplicationContext()* method, we can find a reference to this problem:

> *Context getApplicationContext()*
>
> Return the context of the single, global Application object of the current process. This generally should only be used if you need a Context whose lifecycle is separate from the current context, that is tied to the lifetime of the process rather than the current component.(ANDROID. . . , 2016)

This leads to development errors, since many methods from the platform (specially methods related to UI elements) require a context tied to the current component (Activity, Service), but if an Application context is given, the method will generate an error. For example, to create a simple dialog using an AlertDialog, as depicted in Figure 4.3 in an Android application, we can use the AlertDialog.Builder[2] class that uses the Builder pattern[3] to build a box that overlaps the current activity. The constructor of this class requires a Context reference:

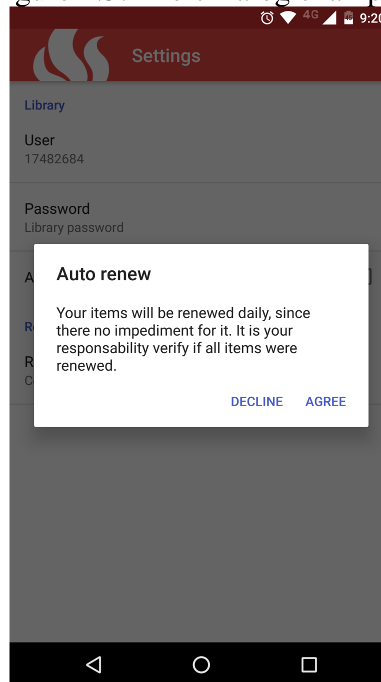
> *AlertDialog.Builder(Context context)*
>
> Creates a builder for an alert dialog that uses the default alert dialog theme. (ANDROID. . . , 2016)

[1]Available at <https://developer.android.com/reference/android/content/ContextWrapper.html>, accessed in May, 2016.

[2]Available at <https://developer.android.com/reference/android/app/AlertDialog.Builder.html>, accessed in May, 2016.

[3]Pattern that separates the construction of a complex object from its representation so that the same construction process can create different representations. (GAMMA, 1995)

Figure 4.3: AlertDialog example.



Extracted from Ufrgs Mobile[4]. Source: Author

However, when the dialog is built and shown, an exception is thrown, whenever the context provided is not an Activity context. This architectural design, joining application and component states in the same type of object, can lead to errors in the app development that may not be simple to detect, since all the objects have the correct type, thus reducing the maintainability of the app.

Another factor compromised in activities and services is the testability. Since those classes are initialized by the Android runtime, we cannot modify their elements before the execution starts. This makes very hard to replace fields with mock objects(BECK, 2003) to perform unit tests, requiring the usage of the dependency injection pattern (SHORE, 2006) to modify those fields.

4.3 Communication with intents

As discussed in Section 2.2.2, Intents are the main communication method between components in Android. This communication is done with the creation of an Intent object, that stores information in a key-value scheme. For an explicit intent, the object is sent to the Android System, that will start the new component with the access to the original intent and all its values. For an implicit intent, the Android System will search which app can handle the created intent and will start the app that also receives this intent.

The organization of messages in intents creates a very flexible architecture, since we can replace a component completely, changing only the sender information in the intent[5]. It is also great for reusability, since we can use intents to get data from other external apps without rewriting functionalities that are already implemented in other apps. For example, if the app needs to take a picture for a simple use case, instead of creating classes and methods to control the camera, we can simply send an Intent to an external camera app, requesting that the app returns the picture when ready. This benefits most of the project quality criteria, such as the overall reusability for using a specialized app already developed, the maintainability and testability since the app does not need to have classes dedicated for this feature. This also benefits product criteria: once the external app has a specific function, its development will be focused on performance and efficiency issues that probably would not be the focus of other apps. Also, the user interface of those external apps tends to be more natural, since they are system apps or apps installed by the user, increasing usability. The interoperability is also increased, since each external app can be specific to the user device, but keeping a common communication interface with app in development.

However, when a component needs to send a large amount of data to another component, the communication by intents can be a problem. The key-value pair of intents is restricted to basic types, such as integers, floats, strings, and objects that implement the Serializable[6] or Parcelable[7] interfaces. This can be a problem for complex structures that cannot implement those methods. In those cases, the app must hold this information in other object or in a static structure to make it accessible by the new component. These structures will increase the complexity of the application and therefore, decrease maintainability.

Another problem created by implicit intents is the lack of self-documentation. Each app will require a specific data format to properly handle incoming data, and it is not possible to document every possible action using implicit intents. This makes the proper use of implicit intents sometimes hard and difficult to detect bugs, as we can see in the example shown in Figure 4.4. To create a new email using an implicit intent, we need to set the destination email, the subject and the body text. To define subject and body text, we set the value in an Intent using the String type. It seems obvious to the programmer to

---

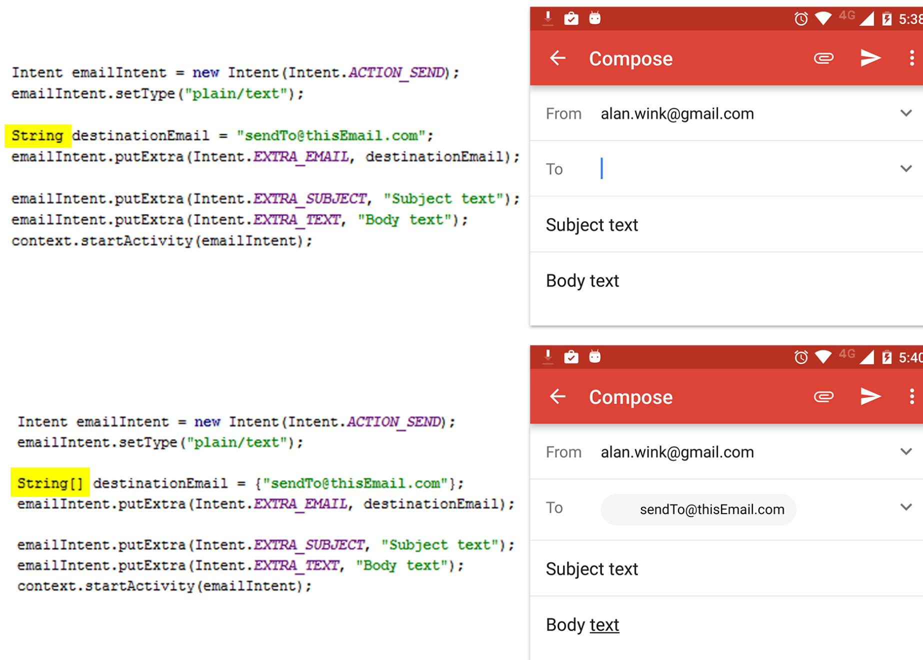
[5]For explicit intents. In the case of implicit intents, the Android System will simply search a new component that can handle the intent and send to it, without any modification.

[6]Available at <https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html>

[7]Available at <https://developer.android.com/reference/android/os/Parcelable.html>

also define the destination email as a String, since the other fields are populated using this type. However, the email application will not read the value, unless it its defined as an array of Strings (since the destination can be multiple emails) and there is no handling for a simple String. The Android Intent documentation[8] is not easy to understand, and most of Intent issues are covered by questions on Android developer websites.

Figure 4.4: An Intent usage example in an email application.



It is possible to verify that the *Intent.EXTRA_EMAIL* is only correctly populated when passing a String array. Source: Author

4.4 Content providers

The content provider is the official component to manage internal and external access to application data. Its structure is created in such a way that it is independent of other application components making it easy to make app data available to other apps. The access to data in a Content Provider is done similary to accesses in a relational database using SQL (Structured Query Language). The content provider also can offer the basic CRUD operations (Create, Retrieve, Update and Delete), creating a very powerful data model.

The Content Provider development is simpler when the data to be accessed is

[8]Available at <https://developer.android.com/reference/android/content/Intent.html>.

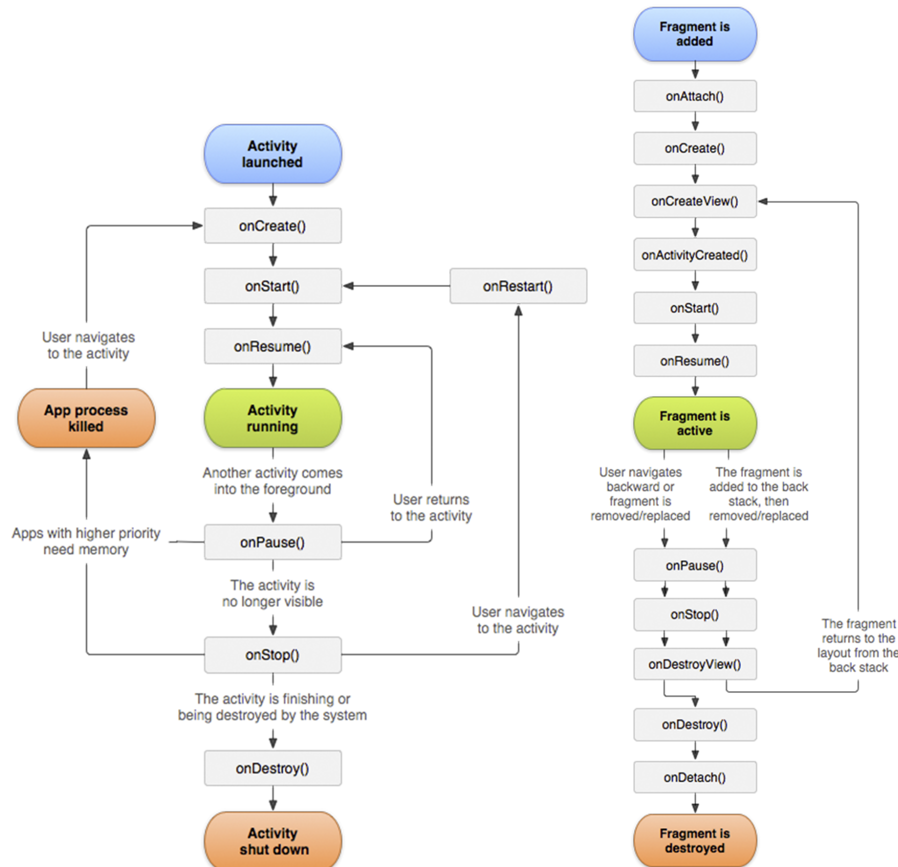
already stored in the SQL database. However, it increases the development complexity for any other type of storage, since the Content Provider needs to send a cursor to the data in the same way relational databases do. This structure drastically decreases the flexibility of the development, since it works better for relational databases, being extremely complex for any other alternative.

Due to its complexity, Content Providers are usually used only when the application needs to share its information with the system or other apps, and other alternatives are used for internal data, as discussed in Section 4.3. The Android system offers access to a large amount of information with Content Providers, such as telephone contacts, calendar events and media available in the device, increasing the overall modularity and increasing the product usability by sharing user information.

4.5 Fragments

As explained in Section 2.2.1, the Fragment component was developed to contain parts of the screen that can be reused in different layouts, depending of several aspects, such as device screen. This approach increases the application modularity and therefore, its reusability. The Fragment component was developed in a similar way to the Activity component, containing several methods to handle different parts of the lifecycle, as depicted in figure 4.5. However, Fragments have more states in its lifecycle, making it harder to understand what parts of the code must be placed in each method, reducing the maintainability.

Figure 4.5: Activity and Fragment lifecycles.



The Fragment lifecycle has more methods, making it difficult to the developer to understand where each part of the code must be and making harder to find bugs, decreasing the application maintainability. Adapted from (ANDROID. . . , 2016).

Similarly to Activities, Fragments cannot be directly mapped to a classical architectural model, as discussed in section 4.2. At first glance, we can understand Fragments as the controller of a view when applying the MVC pattern, since they represent a portion of the screen. However, they keep the same characteristic of Activities, coupling view code - such as layout loading - with business logic, since it still tranfers all platform calls to the Context class.

The way Fragments are instantiated also decreases maintainability and testability. Different than Activities, Fragments must be instantiated in any class, and not by the Android System like Activities. Therefore, we have access to its parameters before showing on the screen. However, all Fragment management is done by a system class called FragmentManager[9], that handles fragment transactions asynchronously. As Ricau (2014)

[9]<https://developer.android.com/reference/android/app/FragmentManager.html>

explains, the fragment transaction is posted at the end of the main thread handler queue, putting the app in an unknown state when receiving multiple clicks before the transaction complete, since the fragment is not ready to operate, leading to bugs that are very hard to identify.

4.6 Custom Views

For many use cases, there is a simpler alternative to fragments that is available since the first Android release: custom views. We can define a custom widget that will be placed on the screen, extending the View[10] class, or the ViewGroup[11] class [12]. We can easily separate different portions of the application screen in custom view groups that will handle the operations related to its views. This way, we can also create another class that will contain business logic, separating view and controller operations in different classes, therefore, making possible to model as MVC and MVP.

4.7 Conclusion

As we could see, the Android platform offers a rich amount of resources to the developer, with an architecture design focused not in traditional architectural patterns. The mobile environment requires from the applications a better usage of its resources, since they are more limited, compared to a desktop environment. The platform was developed focusing in product criteria, creating mechanisms to app developers to build richer solutions when compared to desktop applications by using data from sensor and events sent by broadcast receivers. It also focuses in creating an integrated user experience by using resources from the system or other apps with content providers and explicit intents. The system also focuses on interoperability with the management of resources for different devices specifications, creating an abstraction to the developer about the actual mobile device over which the application is running.

However, some design decisions jeopardize development aspects of the software. Developers need to understand and handle platform characteristics that can lead to a slow software development for larger projects. In the next chapter, we will discuss some design

[10]<https://developer.android.com/reference/android/view/View.html>
[11]<https://developer.android.com/reference/android/view/ViewGroup.html>
[12]ViewGroup is a special view that can host contain other views, creating a hierarchical layout.

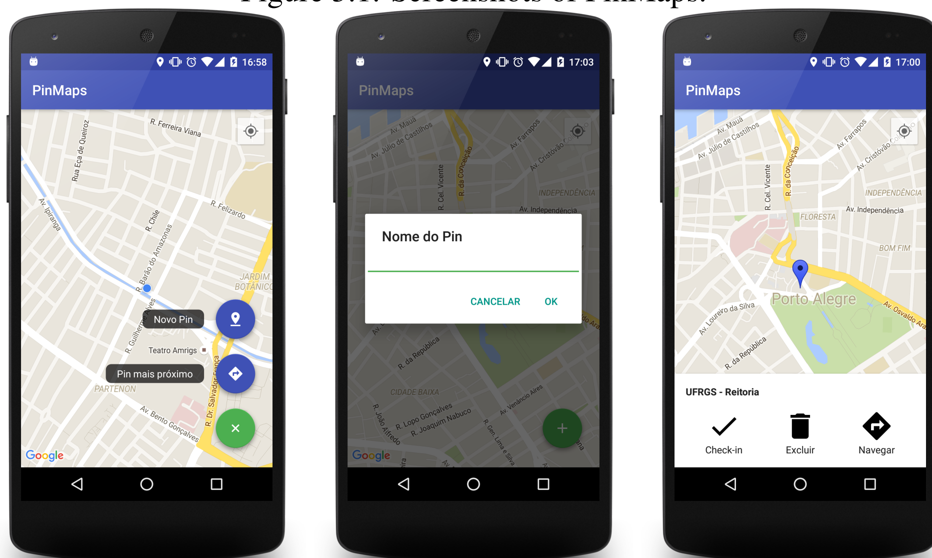
decisions that can be made by the developer to overcome the limitations of the platform, while analyzing the impact on Software Engineering metrics for an example application.

5 INTERNAL QUALITY ANALYSIS FOR A CASE STUDY APP

5.1 App description

To understand the impact of the platform on the application architecture, we designed a simple app called PinMaps. The app helps users to store and locate touristic places, showing to the user the closest stored place, based on his/her location. The application makes use of the GPS sensor, network and database provided by the Android platform. The user interface of the application can be seen in Figure 5.1.

Figure 5.1: Screenshots of PinMaps.



Source: Author

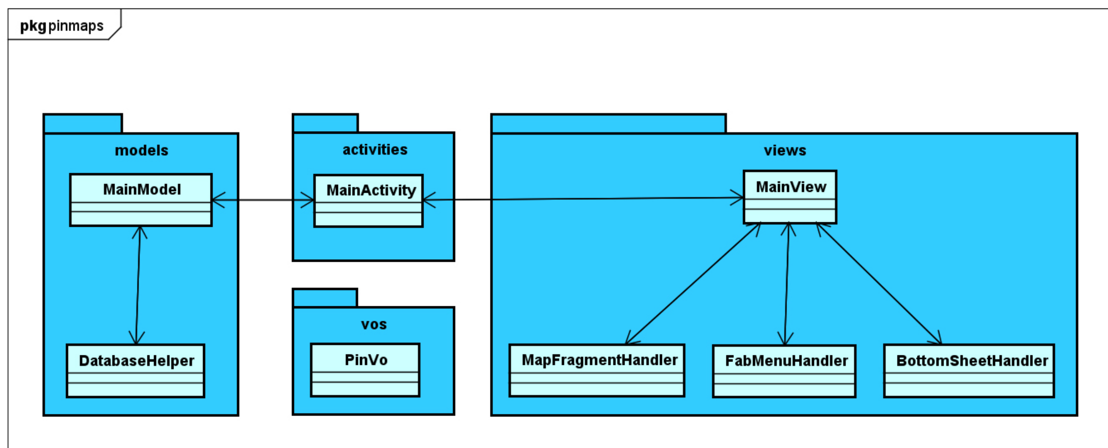
5.2 Developed versions

The app was developed using an adapted version of the MVC pattern, proposed by Musselwhite (2011). In this architecture, all operations related to the user interface are implemented in Custom Views, as discussed in Section 4.6. This structure leaves the Activity with only methods that are not related to user interface, such as sensor readings or lifecycle methods, as depicted in Figure 5.2. We also decoupled methods in the view layer, isolating different elements of the screen in different classes, following the principles of Single Responsibility and Open Closed (MARTIN,).

We developed four additional versions of the application, in order to analyze the impact of the application architecture in software engineering good practices. In each new

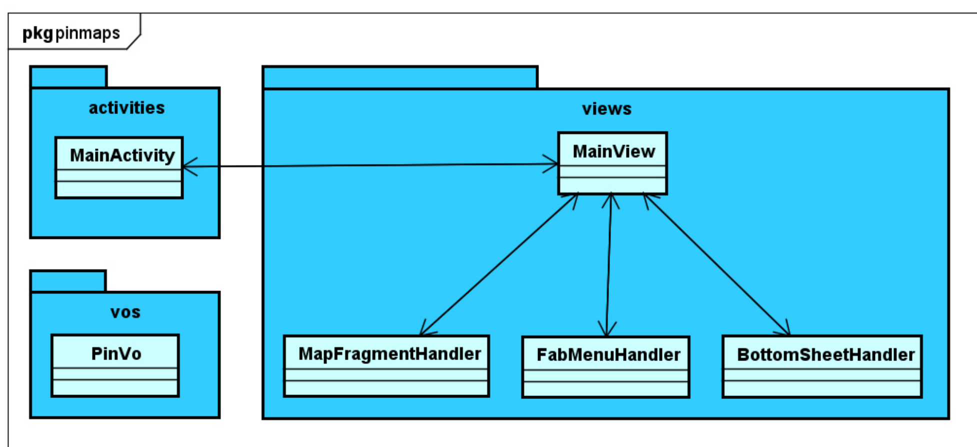
version, modules were collapsed to reduce the overall cohesion, as depicted in Figures 5.3, 5.4 and 5.5, respectively. In the last version, the app is implemented using a single Java file, as we can see in Figure 5.6.

Figure 5.2: Version 1 - Original model for PinMaps.



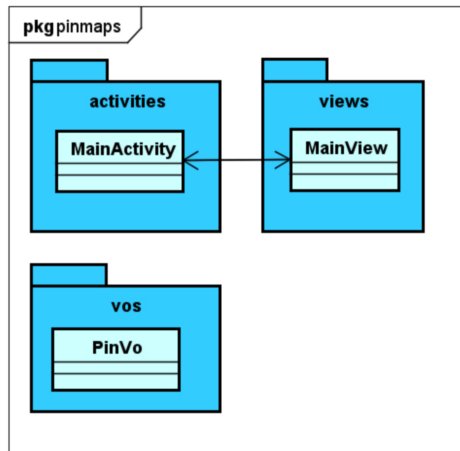
Source: Author

Figure 5.3: Version 2 - Model and controller coupling.



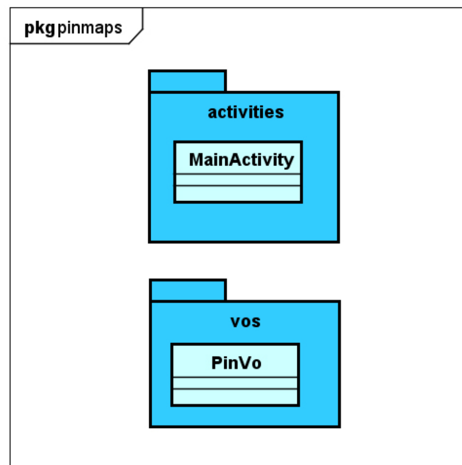
Source: Author

Figure 5.4: Version 3 - View coupling, reducing all views to one class.



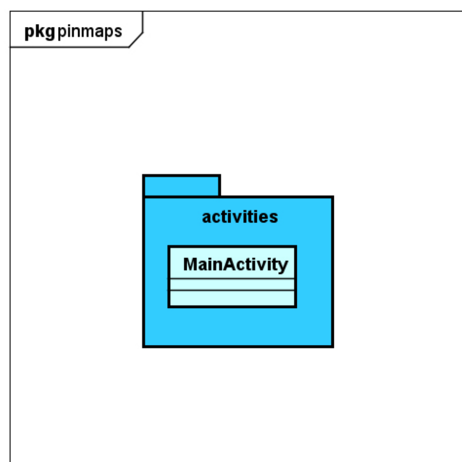
Source: Author

Figure 5.5: Version 4 - View and Controller coupling.



Source: Author

Figure 5.6: Version 5 - Full coupling. Removal of all unecessary methods by replacing the method call with the actual code (inline expansion).



Source: Author

5.3 Extracted metrics

For each version of the app, design metrics were collected for further analysis. The following metrics were considered:

- *C*: Classes. The number of classes in the project.
- *CF*: Coupling factor. The proportion of classes that are used by a class, in average.
- *LOC*: Lines of code. The number of lines of code in the entire project.
- *METH*: Methods. The number of methods in all the classes of the project.
- *v(G)avg*: Average cyclomatic complexity. Average cyclomatic complexity of all non-abstract methods of the project.

To extract these metrics, we used the plugin MetricsReloaded[1], which is compatible with Android Studio, the oficial IDE (Integrated Development Environment) for Android development.

5.4 Results

The metrics results are presented in Table 5.1 which shows that the most relevant change to the application metrics is in the coupling factor of version 5. This version is extremely hard to maintain, being the worst version for all project factors discussed on Section 3.2.4 (Flexibility, Reusability, Maintainability and Testability).

Table 5.1: Extracted metrics for each version of the app.

Version	C	CF	LOC	METH	v(G)avg
1	23	56.41%	1681	109	1.42
2	21	65.45%	1584	98	1.45
3	17	60.71%	1466	83	1.48
4	16	61.90%	1439	81	1.49
5	17	86.67%	1218	14	4.54

Source: Author.

We discussed in Chapter 2 the elements that the platform provides to the developer to build an application. Since PinMaps has only one screen where the user interacts with the app, it seems reasonable to develop the application in a single Activity file from a

[1] Available at <https://github.com/BasLeijdekkers/MetricsReloaded>.

platform perspective. This architecture is represented in our analysis in version 4, where all the business logic is handled inside *MainActivity* class.

Analyzing the extracted metrics, we can see there is a small difference in the metrics from version 1 to version 4. However, version 1 leads to an application much better structured, increasing its project quality factors, which is essential for a production application that will be maintained over a period of time, fixing errors and adding new functionalities through updates.

There are many other approaches to build more maintainable Android applications. In the next chapter, we will discuss elements outside the Android platform that help developers to build high-quality apps, not only from a developer perspective, but also from a user perspective, using libraries that increase the Usability and Interoperability of the app.

6 QUALITY ELEMENTS OUTSIDE THE ANDROID PLATFORM

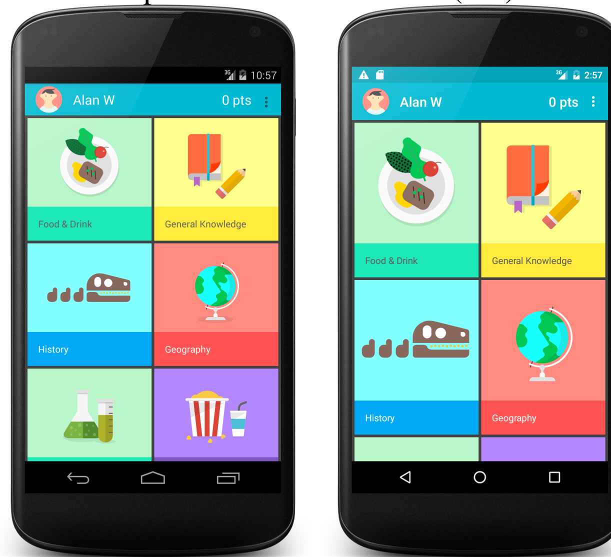
Some of the Android platform problems discussed on Chapter 4 can be reduced using techniques not provided by the platform. That are plenty of techniques and libraries created by Google and other Android developers that help to increase the quality of the final application. In this chapter, we will discuss some of these techniques and their impact on the quality of the final application.

6.1 Backwards compatibility support

With the continuous release of new Android APIs, the mobile developer has the opportunity to use new features. However, these features would require from the programmer to stop supporting older APIs in order to support new features, leading to an Interoperability problem. In order to encourage developers to use new features without sacrificing support for older APIs, Google released the Android Support Library. The library is actually a collection of small libraries that have four objectives: Backwards support of framework elements, UI implementations following the Android design manual, support for different form factors (such as TVs and Wearables) and utility classes(SUPPORT. . . , 2016). In Figure 6.1, the app Topeka[1] uses the Support Library to create an user interface using elements released in API 21 (such as the Toolbar[2] class), but keeping compatibility with older APIs.

[1] Available at <https://github.com/googlesamples/android-topeka>
[2] Available at <https://developer.android.com/reference/android/widget/Toolbar.html>

Figure 6.1: Screenshots of Topeka on API version 19 (left) and API version 23 (right).



Source: Author

The Support Library is becoming the common way to create projects for Android. By default, the Android Studio creates new projects using the Support Library elements such as AppCompatActivity[3] (the equivalent of Activity in the platform) or its own implementation of the Fragment. The Support Library usage increases Interoperability by supporting devices with older API versions, as well as Usability, since new UI elements that are available in recent APIs are also available for older APIs. From a developer perspective, it makes the application easier to be developed, since there will be no extra code to handle with older devices, as well as with new APIs to be released, since the Support Library will handle any differences for the new platform, increasing the Maintainability and Reusability of the application.

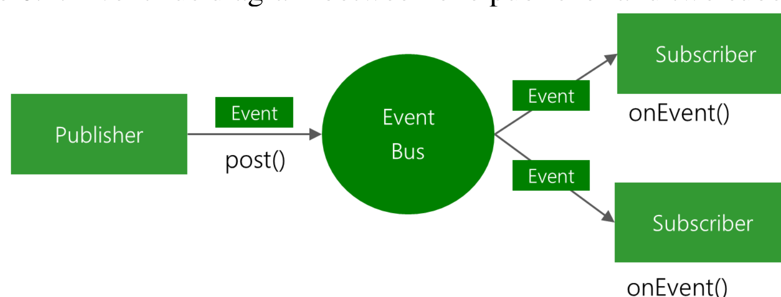
6.2 Communication alternatives

As discussed in Section 4.3, one of the problems of Intents is passing complex information, requiring serialization and deserialization. Also, for other elements, the communication can be different. For example, a Service running without an Activity connected will require to send an event to a BroadcastReceiver to notify the application. For fragments, Activities can pass arguments by Intents, or simply by Java method calls. These approaches lead to strong dependencies between elements, reducing its modularity.

[3]Full documentation available at <https://developer.android.com/reference/android/support/v7/app/AppCompatActivity.html>

One approach used to reduce this dependency is the event bus model. In this model, there are two roles: publishers who are responsible to send event and subscribers that are responsible to respond to these events. The event becomes the intermediary element that isolates publishers and subscribers. This model also helps to solve the problem of complex objects, since publishers can send Java objects in their events. There are two main libraries supported by the community to implement this model in Java: EventBus[4] and RxJava[5]. In Figure 6.2 we can see a diagram of the EventBus library. Since the publisher is not directly connected to the subscriber, there is no coupling between sender and receiver, creating a flexible communication system.

Figure 6.2: EventBus diagram between one publisher and two subscribers.



Available at <http://greenrobot.org/eventbus/>.

6.3 Storage elements

As discussed in Section 4.4, there are other ways to store app data in an Android application. The SQL database provided by the platform can be a problem, since it requires the developer handles the data conversion from Java objects to rows in a SQL table. In order to increase time-to-market, many developers prefer to use libraries that handle data storage in an Android application rather than creating its own SQL database and classes to handle the data. One alternative to the SQL storage is the Realm Database[6]. The Realm database is accessed directly by Java methods and uses Java objects as input, without requiring a conversion class from the programmer. This increases the Maintainability of the application, since the conversion is responsibility of the database, and not from the programmer. In Figure 6.3, there is an example of an object creation and storage using a simple Java object, as well as a retrieval using simple Java methods.

[4] Available at <https://github.com/greenrobot/EventBus>
[5] Available at <https://github.com/ReactiveX/RxJava>
[6] Available at <https://realm.io/>, source code available at <https://github.com/realm>

Figure 6.3: Realm database example.

```
public class Dog extends RealmObject {
    public String name;
    public int age;
}

Dog dog = new Dog();
dog.name = "Rex";
dog.age = 1;

Realm realm = Realm.getDefaultInstance();
realm.beginTransaction();
realm.copyToRealm(dog)
realm.commitTransaction();

RealmResults<Dog> pups = realm.where(Dog.class)
                              .lessThan("age", 2)
                              .findAll();
```

Adapted from <https://realm.io/>.

Other alternative widely used is the Parse Platform[7]. The Parse Platform is commonly used to store and access data online, requiring little effort from the developer to retrieve and send data, that is handled by the Parse SDK (Standard Development Kit) on Android. The SDK creates an interface very similar to local data access, and also handles no-connectivity issues, caching and syncing data when the network connection is reestablished.

6.4 Testing elements

One of the factors to achieve better software quality is the application Testability, as discussed in Section 3.2.4. To help developers achieve this goal, Google grouped their libraries related to tests in a single library called Android Testing Support Library. This library provides an extensive framework to test Android apps. The library has three main elements: the AndroidJUnitRunner, a unit test runner compatible with Junit 4[8], the Espresso framework, designed to create UI tests and the UI Automator, responsible for creating cross-app[9] UI tests(TESTING. . . , 2016).

The Espresso framework is especially important to test applications in multiple

[7] Available at <https://github.com/ParsePlatform>

[8] A framework created to support unit testing in Java. Available at <http://junit.org/junit4/>.

[9] Tests that require to open multiple apps to simulate a use-case, like testing Implicit Intents as discussed in Section 4.3.

devices. The framework is designed to require information about the application UI, and not about specific device characteristics, such as screen size or orientation. This makes the framework very flexible and capable to be used in testing platforms like Google Cloud Test Lab[10], where developers send their applications and Espresso tests to run with different physical devices.

Another approach widely used to find errors is the usage of crash trackers. Since developers do not have access to all devices their applications can run and the distribution system by app stores makes it easy to update applications, dealing with errors after a release is relatively easy. To find these errors, developers attach app trackers, like Google Analytics[11] or Crashlytics[12] that are capable of sending the crash information to the developer, as well as device information, in order to find and solve the error. This approach helps in the Maintainability of the application, and in some way its Testability, since errors can be detected and solved.

[10] Available at <https://developers.google.com/cloud-test-lab/>
[11] Available at <https://developers.google.com/analytics/>
[12] Available at <https://try.crashlytics.com/>

7 CONCLUSION

In this study, we presented the Android platform, analyzing its structure from a developer perspective. Considering Software Engineering aspects, the platform has some problems concerning specially modularity issues. The mobile market is increasing and applications are becoming more complex. Therefore, the discussion about Software Engineering techniques is becoming even more important, since apps must be constantly updated to keep relevant in the market. To ensure quality, app developers must rely in architectural models for the Android platform, which is something not frequently discussed. Only a few works have been dedicated to the Android application architecture, while the Android community identifies an architecture as an important part of successful system design and development (SOKOLOVA et al., 2014).

The Android community has created libraries and components to make developers build better apps using known architectural models, as well as simplifying some elements that are complex in the platform, such as databases and internal communication. Different from the iOS platform, that enforces the MVC pattern (APPLE DEVELOPERS, 2015), the Android platform does not have a defined model, opening a discussion to find good architectures and techniques to the mobile environment.

Some decisions made in the Android platform can be explained in terms of performance. The model based in Activities to each app screen is extremely useful for saving battery, since Activities that are not shown can be stored and removed from memory without user noticing. There are also evidences that components outside the platform have a better performance than platform components. For example, the Realm database discussed in Section 6.3 presents some studies showing it has a better performance in the iOS platform than the native alternative (DOBRINCU, 2014), and some studies are also available for the Android version[1]. The impact of Software Engineering techniques and alternative components in mobile performance can be analyzed for a future work.

[1] Available at <https://github.com/klinker41/android-realm-performance>

REFERENCES

AHONEN, T. T. **Smartphone Wars: Q3 Scorecard - All market shares, Top 10 brands, OS platforms, Installed base**. 2015. Available from Internet: <http://communities-dominate.blogs.com/brands/2015/10/smartphone-wars-q3-scorecard-all-market-shares-top-10-brands-os-platforms-installed-base.html>.

ANDROID Developers Reference. 2016. Available from Internet: <http://developer.android.com/reference/packages.html>.

APP Manifest. 2016. Available from Internet: <http://developer.android.com/guide/topics/manifest/manifest-intro.html>.

APPBRAIN. **Current number of Android apps on Google Play**. 2016. Available from Internet: <http://www.appbrain.com/stats>.

APPLE DEVELOPERS. **Model-View-Controller**. 2015. Available from Internet: <https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>.

APPLICATION Fundamentals. 2016. Available from Internet: <http://developer.android.com/guide/components/fundamentals.html>.

BECK, K. **Test-driven development: by example**. [S.l.]: Addison-Wesley Professional, 2003.

DOBRINCU, S. **5 Reasons Why You Should Choose Realm Over CoreData/SQLite**. 2014. Available from Internet: <http://sebastiandobrincu.com/blog/5-reasons-why-you-should-choose-realm-over-coredata>.

FILTERS on Google Play. 2016. Available from Internet: <https://developer.android.com/google/play/filters.html>.

FRANKE, D.; KOWALEWSKI, S.; WEISE, C. A mobile software quality model. In: IEEE. **Quality Software (QSIC), 2012 12th International Conference on**. [S.l.], 2012. p. 154–157.

GAMMA, E. **Design patterns: elements of reusable object-oriented software**. [S.l.]: Pearson Education India, 1995.

GRODEN, C. **Google: Mobile searches surpass desktop searches worldwide**. 2015. Available from Internet: <http://fortune.com/2015/10/08/google-mobile-searches-surpass-desktop-searches-worldwide>.

ISO/IEC. **Software Engineering - Product Quality, ISO/IEC 9126-1**. [S.l.], 2001.

ISO/IEC. **ISO/IEC 25010:2011 - Systems and Software Engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models**. [S.l.], 2011.

MARINHO, E. H.; RESENDE, R. F. Quality factors in development best practices for mobile applications. In: **Computational Science and Its Applications–ICCSA 2012**. [S.l.]: Springer, 2012. p. 632–645.

MARTIN, R. C. **The Principles of OOD**. Available from Internet: <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>.

MCCALL, J. A.; RICHARDS, P. K.; WALTERS, G. F. **Factors in software quality. volume i. concepts and definitions of software quality**. [S.l.], 1977.

MUSSELWHITE, J. **Android Architecture**. 2011. Available from Internet: <http://www.therealjoshua.com/2011/11/android-architecture-part-1-intro/>.

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. [S.l.]: Palgrave Macmillan, 2005.

RICAU, P.-Y. **Advocating Against Android Fragments**. 2014. Available from Internet: <https://corner.squareup.com/2014/10/advocating-against-android-fragments.html>.

SHORE, J. **Dependency Injection Demystified**. 2006. Available from Internet: <http://www.jamesshore.com/Blog/Dependency-Injection-Demystified.html>.

SOKOLOVA, K. et al. Towards high quality mobile applications: Android passive mvc architecture. **International Journal on Advances in Software**, Citeseer, v. 7, p. 123–138, 2014.

SUPPORT Library. 2016. Available from Internet: <https://developer.android.com/topic/libraries/support-library/index.html>.

TAKAHASHI, D. **Google releases details on Android Market launch**. 2008. Available from Internet: <http://venturebeat.com/2008/10/22/google-releases-details-on-android-market-launch/>.

TEAM, M. P. . P. **Microsoft Application Architecture Guide**. Microsoft Press, 2009. (Microsoft Press Series). ISBN 9780735627109. Available from Internet: <https://books.google.com.br/books?id=qjxqPgAACAAJ>.

TESTING Support Library. 2016. Available from Internet: <https://developer.android.com/topic/libraries/testing-support-library/index.html>.

VINCENT, J. **Android is now used by 1.4 billion people**. 2015. Available from Internet: <http://www.theverge.com/2015/9/29/9409071/google-android-stats-users-downloads-sales>.

WASSERMAN, A. I. Software engineering issues for mobile application development. In: ACM. **Proceedings of the FSE/SDP workshop on Future of software engineering research**. [S.l.], 2010. p. 397–400.