POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Kotlin and Android applications: diffusion and adoption of characteristic constructs

Supervisors

Prof. LUCA ARDITO

Prof. MARCO TORCHIANO

Candidate Hussein ZAYAT

MARCH 2020

Abstract

Context: In October 2017, Google introduced Kotlin as a first class programming language for android applications. Kotlin is an Object oriented programming compatible with java, with the addition of some practices that helps to reduce number of code lines, exceptions and other issues that face developers. Many mobile applications started a transition by changing the base code from java to Kotlin.

Goal: The main objective of the article is to study the impact of Kotlin on open source android applications on GitHub. To analyze the impact the paper were divided into three parts. Firstly, it aims to study the interactions with Kotlin applications on GitHub(Issues,Contributors,Stars). Secondly, the usage of Kotlin practices inside applications. Finally, It aims to study the code quality on a sample set of releases.

Method: First we started by fetching GitHub API's to clean and collect data about applications and then download only repositories that represent applications. We defined a method that measures the percentage of adoption of Kotlin on all applications to compute the metrics needed to answer diffusion research question. Then on a set of tagged releases of the applications we measured metrics needed for the evolution research question. After that we parsed the source code of the last version of applications(Master branch on GitHub), searching Kotlin development practices using regular expression to study the usage of Kotlin features after the transition. Finally, we defined a set of code smells that are common between Kotlin and Java, and using the Lint static analysis tool provided by android studio, we calculated the Code Quality of applications on two different releases.

Results: On the data-set of 33,267 open source applications, we found that 33,212 (99.9%) applications featured at least one Kotlin file, while 32,877 (98.8%) applications had a majority of Kotlin files with respect to java. 96.5% of the lines of code of all applications was written in Kotlin and 94.8% of files are ".kt" files. We found using Code Inspection that 5,022 applications (15%) are not using any of the Kotlin development practices. We considered 458 applications with tagged releases before and after October 2017 to answer the evolution research question. We found that only 19/458 featured a minority of Kotlin, While 252 releases featured majority of Kotlin with the existence of java in the production code. While the rest of releases(187) are fully converted to Kotlin. After inspecting the source code of applications searching for Kotlin development practices, we found that unsafe casting is used in 71% of applications, while safe casting is used only in 11%. Null Assertion Operator(!!) is used in 56% of the applications and safe calls are used in 70.4% of applications. Data classes and long argument lists were found

respectively in 49% and 15% of the applications. Finally 54% of the applications declared variables that accept null as a value.

Conclusion: A big set of apps is still in the phase of transition since Java is still used in the production code of the last release and the percentage of applications with majority of Kotlin locs over the production code is increasing. Comparing these results with the usage of Kotlin development practices, we concluded that applications made a huge transition without taking advantage of development practices that may be because of using some tools to convert the base-code.

Acknowledgements

"Surround yourself with those conducive to you being your highest self." by Posey is a quote that I follow in every step of my life. Writing this thesis has been fascinating and extremely rewarding. I would like to thank everyone who in a one way or another contributed in the completion of this thesis. This accomplishment was harder to achieve without them.

I would first like to thank my thesis advisor Prof. Riccardo Coppola at Politecnico di Torino. The door to Prof. Coppola office was always open whenever I had a question about my research or writing. Without your supervision, I would not have completed this thesis.

I would also like to thank Prof. Luca Ardito and Prof. Marco Torchiano. I respectfully thank you for the trust and for giving me the chance to study under your supervision.

Some special words of gratitude go to my friends Mohamad, Hassan and Abess. Who have always been a major source of support when things would get a bit discouraging. Thank you guys.

I would like very much to mention Walter Gamba, Danielle Barbaro, Piergiorgio Barra and all my colleagues in Iakta. Thank you for the support and time you gave me during working on my thesis.

Finally, I must express my very profound gratitude to my parents, my brothers and my sister for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. This thesis is dedicated to them.

Thanks to my uncles, cousins, friends and university colleagues, for all the help and support.

Table of Contents

List of Tables

List of Figures

Acronyms

OS
: Operating System

IDE
: Integrated Development Environment

ML
: Micro Edition

JVM
: Java Virtual Machine

API
: Application Programming Interface

NPE
: Null Pointer Exception

KRL
: Kotlin Relative Lines of code

KRF
: Kotlin Relative Files

KFPR
: Kotlin Featuring Projects Ratio

KMPR
: Kotlin-Majority Projects Ratio

KAR

Kotlin Adoption Relative Releases

KMR

Kotlin Majority Relative Releases

KOR

Only Kotlin Relative releases

CC

McCabe's Cyclomatic Complexity

WMC

Weighted Method Count

RLOC

Relative number Lines Of Code

STAT

Count of Statements inside a method

Chapter 1

Introduction

Kotlin is a new programming language that represents an alternative to Java, it was introduced in October 2017 as a first class programming language for Android applications. Almost all Kotlin developers (92%) were using Java before they started using Kotlin. Most of them (86% of all Kotlin users) still continue to use Java.[1]

The study aims to evaluate the transition to Kotlin programming language of a large list of open-source Android applications over their lifespan and show the impact of this transaction on the success of Android applications.

1.1 Android Operating System

1.1.1 Overview

Mobile phones were considered as the fastest growing man-made phenomenon ever. While the number of mobile phones is surpassing the number of people in the world and the big impact of the smartphone evolution on the users.

The Android Operating system was found in October 2003. Few months later android gears decided to use the operating system inside mobile phones, and in 2005 Android was acquired by Google. Then in November 2008, the first android beta was released and it was followed by the first commercial mobile operating system version release (Android 1.0) in September 2008. Since that release , Android had seen several updates to its operating system. Android used code names starting from version number 1.5 in April 2009 (Cupcake) until August 2019 when Google announced that the numerical ordering will be used for future versions staring from Android 10 (Api 29).[2]

1.1.2 Statistics and Facts

Android is the most used operating system around the world and according to the announcement in May 2019, Android operating system was installed on 2.5 billion active devices, those devices were produced by more than 180 hardware manufacturers[3]. According to market share analysis during the final quarter of 2019, we have discovered that Android mobile phones are better than other operating system phones. Android has more than 75 percent of the market share and is the most famous smartphone OS on the planet today.[4]

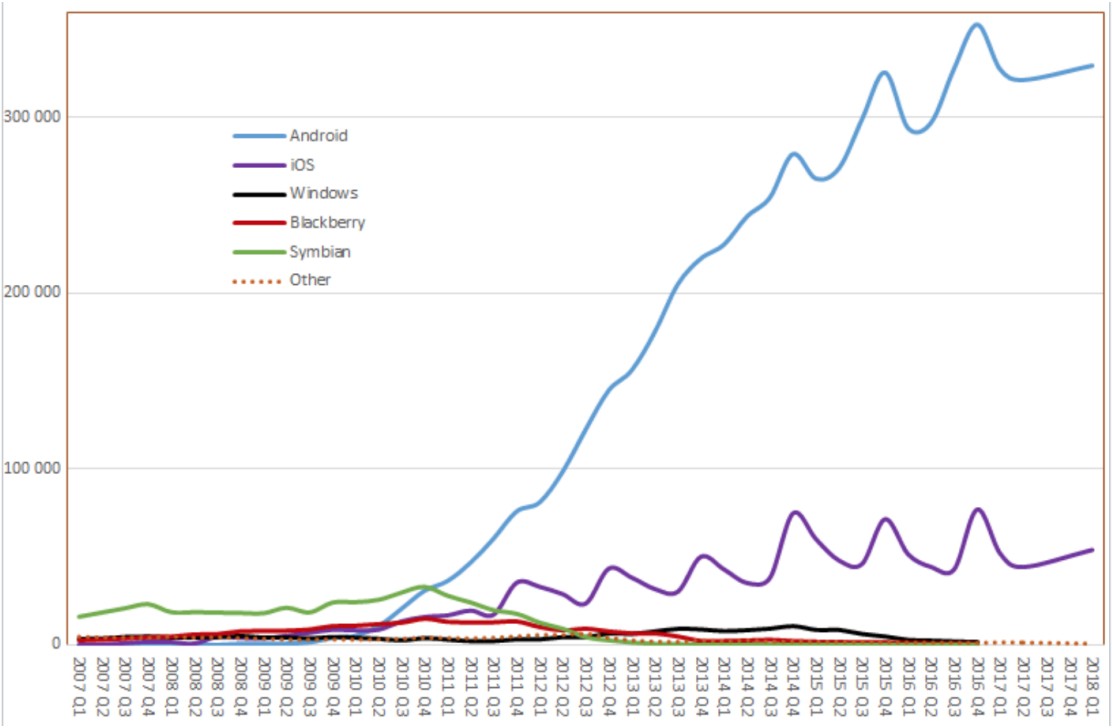

Figure 1.1: World Wide Smartphone Sales
[4]

Upon the accomplishment over smartphones and expanding fame of Android devices directly affects the app store Google Play, which was first presented under the name of Android Market. It is presently the greatest application store on the planet, and had 3.6 million applications accessible to download in March 2018.[5]

1.2 Android Mobile Applications

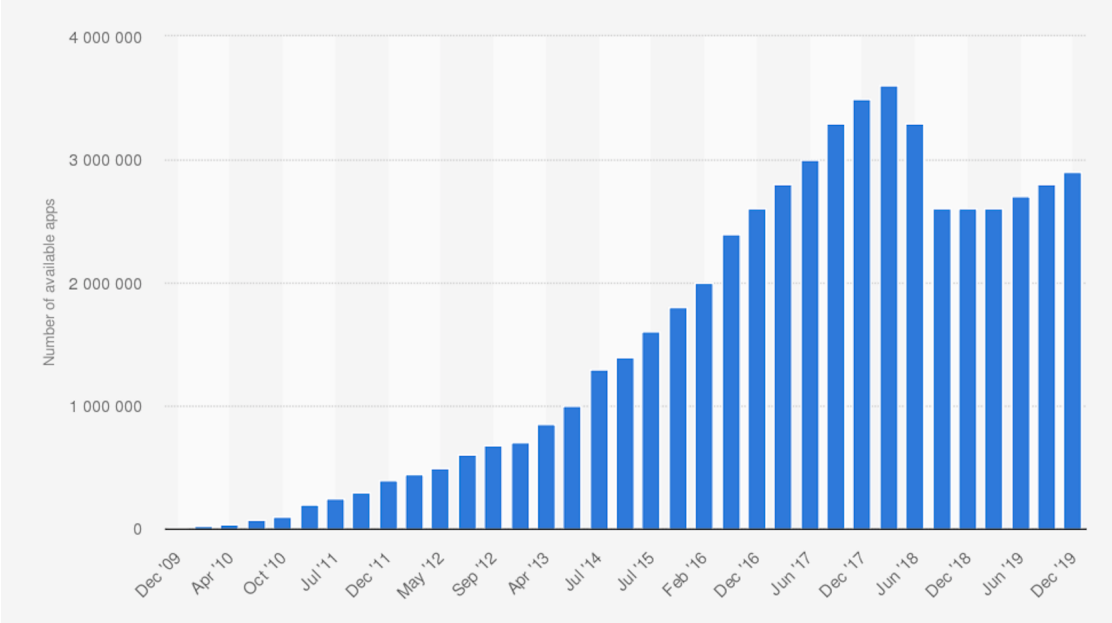

Figure 1.2: Number of available applications in the Google Play Store Between Dec '09 and Dec '19

[6]

1.2.1 Innovation and New Experiences

With Android 10 developers can exploit the most recent programming innovations to build new applications features with an amazing user experience:[7]

- **Foldables**: Android 10 extends multitasking across app windows and provides screen continuity to maintain your app state as the device folds or unfolds.
- **Smart Reply in notifications**: Android 10 uses on-device ML to suggest contextual actions in notifications
- **Dark Theme**: Developers can build now a custom dark theme for applications or opt-in to a new Force Dark feature that lets the system dynamically create a dark version for an existing theme.
- **5G Networks**:For a faster speed and lower latency, Android 10 adds platform support for 5G. Now developers using connectivity API's , can take advantage of these improvements that provides a faster and immersive experiences to application users.

1.2.2 Android Open-Source Applications

Android is an open source working framework, clients can install third party applications easily from markets(even from an untrusted source). Because of this, it has some constraints which lead to malware attacks and viruses. Several app markets are available for Android developers to release or sell their apps, such as the official Google Play store, the Amazon AppStore, F-Droid, GetJar, itch.io . F-Droid2 is a repository of free and open-source apps for Android devices, of which both the APK with the compiled code and a source tarball are provided.

1.2.3 Development Tools

Android studio as the official integrated development environment appears to top the list of favored tools for Android developers. Back in 2013, Google created Android Studio, it replaced Eclipse Android Development Tools as the essential IDE for native Android applications development.[8]

Android Studio is a free download IDE and its supported by the community of Android developers. It provides a list of useful and easy to use tools for developers such as code editing, debugging and testing tools. However, Android developers can produce applications much faster and can freely choose between integrated development environments.

Some of the favored tools for android developers:

- Android Debug Bridge(ADB)
- Android Virtual Device (AVD) Manager
- Eclipse
- IntelliJ IDEA

1.2.4 F-Droid

F-Droid is an installable catalogue of FOSS (Free and Open Source Software) applications for the Android platform. The client makes it simpler to browse, install, and monitor of updates on your device.[9]

1.3 Applications Development Languages

Mobile applications may seem easy to be developed. However, The choice of the best programming language to a develop mobile applications is not that easy, as it characterises the general user experience, application response time and ease of use of the application to the purchasers. This is the reason why some programming

languages consistently come first from others in competition of the best coding language for mobile applications.

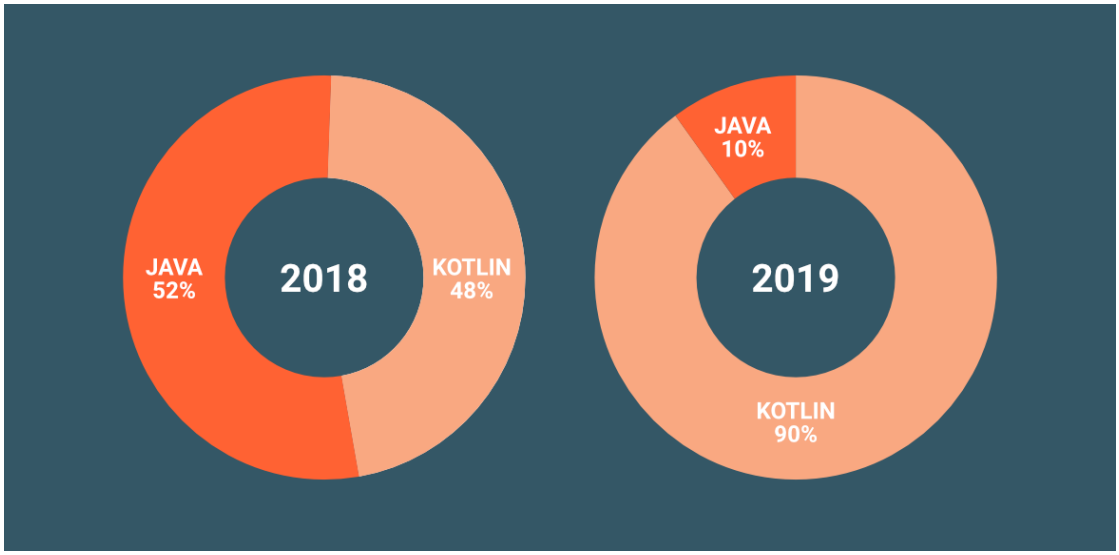

Figure 1.3: Comparison of Java and Kotlin Android Apps as of 2018-2019 [10]

1.3.1 Java

Java is a programming language first released by Sun Micro-systems back in 1995. It can be found on many different types of devices from smart-phones, to mainframe computers. Java depends on a "Virtual machine" which comprehends an intermediate design called java byte-code [11]. Java is one of the preferred languages for Android development:

- It's a well known language amongst the developers with a large number of development tools.
- It's not required to recompile the code for every device the code it used in because it runs in a virtual machine
- Speed is an issue for JAVA, yet its popularity and advantages overweight's over the speed.
- Several mobile phones already used Java ME, so Java was known in the industry.

1.3.2 Kotlin

Overview

Kotlin was introduced with a main objective "Android applications development". In May 2019, Google announced that the Kotlin programming language is now its preferred language for Android app developers.[12] Kotlin attracted many developers because of it's simple syntax and main focus on mobile development and mainly because of it's compatibility with Java. The ease of transition from Java to Kotlin was the key of success with mobile development.

Kotlin Main Features

Kotlin features were interesting to many mobile developers. However, the main features used to advertise the conversion to Kotlin are:[13]

- **Concise**: Drastically reduce the amount of boilerplate code.
- **Safe**: Avoid entire classes of errors such as null pointer exceptions.
- **Interoperable**: Leverage existing libraries for the JVM, Android, and the browser.
- **Tool-friendly**: Choose any Java IDE or build from the command line.

Kotlin Statistics

We can see in figure 1.4 the distribution of Kotlin applications. According to a study done by researchers at the University of Valenciennes, 58 out of 109 (53.21%) applications have at least 80% of Kotlin code. Moreover, 21 out of 925 (19.27%) applications, have less than 10% of Kotlin code.[14]

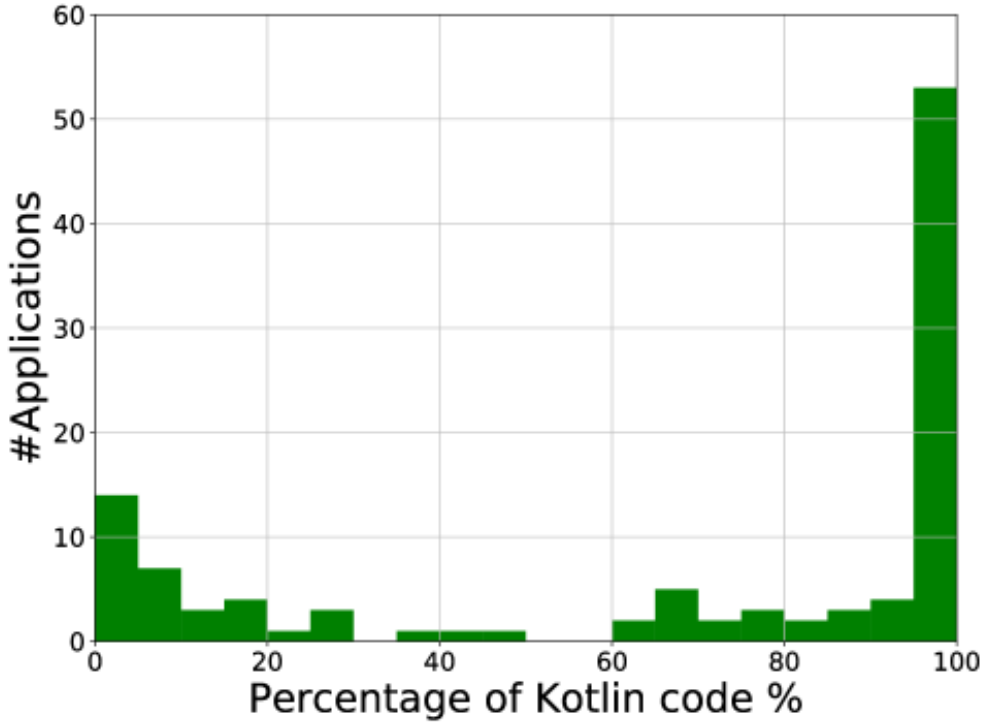

Figure 1.4: Distribution of applications according their percentage of Kotlin code. 53% of Kotlin applications have more than 80% of source code written in Kotlin. [15]

1.4 Github Platform

1.4.1 Overview

GitHub offers free accounts that are commonly used to host open source projects. As of January 2020, GitHub reports having over 38 million users[16] and more than 100 million repositories[17] (including at least 30 million public repositories)[18], making it the largest host of source code in the world.[**github:largest-host**]

1.4.2 Github API

GitHub offers an API for developers that want to develop applications targeting their interface. It provides a clear documentation for developers to access public repositories Api's easily. JSON is used to send and recieve data, and all API access is over HTTPS and accessed from https://api.github.com[17].

1.4.3 API Rate Limiting

Header Name	Description
`X-RateLimit-Limit`	The maximum number of requests you're permitted to make per hour.
`X-RateLimit-Remaining`	The number of requests remaining in the current rate limit window.
`X-RateLimit-Reset`	The time at which the current rate limit window resets in UTC epoch seconds.

Figure 1.5: Github rate Limit Documentation
[19]

Unauthenticated requests

Unauthenticated requests are not associated with the user making the request, but to the originated IP address, so a rate limit allows up to 60 requests per hour which might not be enough for developers using the Api with applications. [19]

```
curl -i https://api.github.com/users/octocat
HTTP/1.1 200 OK
Date: Mon, 01 Jul 2013 17:27:06 GMT
Status: 200 OK
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 56
X-RateLimit-Reset: 1372700873
```

Figure 1.6: Unauthenticated rate Limit response
[19]

Authenticated requests

For API requests using Basic Authentication or OAuth, Users can benefit from the higher rate limit up to 5000 requests per hour, creating an application and then passing your app's client ID and secret as part of the query string on each request of the GitHub API. [19]

```
curl -i 'https://api.github.com/users/whatever?client_id=xxxx&client_secret=yyyy'
HTTP/1.1 200 OK
Date: Mon, 01 Jul 2013 17:27:06 GMT
Status: 200 OK
X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4966
X-RateLimit-Reset: 1372700873
```

Figure 1.7: Authenticated rate Limit response [19]

Abuse rate limits

In order to provide quality service on GitHub, Some actions may result in abuse rate limiting as to provide quality service on GitHub.[19]

- Using the API to rapidly create content.
- Poll aggressively instead of using webhooks.
- Make multiple concurrent requests.
- Repeatedly request data that is computationally expensive.

Chapter 2

Mining of Kotlin Android projects From Github

Our study aims to compare the projects before and after the transition to Kotlin and the impact on the success of Android applications. The first step of the procedure was to mine all GitHub repositories that belong to Android with the language specified as "Kotlin". Then we collect all the necessary information related to our study. Choosing GitHub repositories as a reference was because of the big number of projects that may help to have more accurate results. To start with the data mining, We developed a set of Java classes that requests data from GitHub APIs, and in order to improve the quality of the statistics and to reduce the percentage of error, We collected all the data related to the number of releases, issues, stars and the number of contributors for each application related to our study. Figure 2.1 shows the structure of the database table were the information about applications was stored.

ProjectsInfo		CREATE TABLE ProjectsInfo(Id INTEGER NOT NULL
Id	INTEGER	"Id" INTEGER NOT NULL
OwnerName	TEXT	"OwnerName" TEXT NOT NULL
ProjectName	TEXT	"ProjectName" TEXT NOT NULL
Stars	INTEGER	"Stars" INTEGER
Size	INTEGER	"Size" INTEGER
Contributors	INTEGER	"Contributors" INTEGER
Releases	INTEGER	"Releases" INTEGER
Issues	INTEGER	"Issues" INTEGER
Commits	INTEGER	"Commits" INTEGER
done	BOOL	"done" BOOL

Figure 2.1: Database table structure for GitHub Projects information

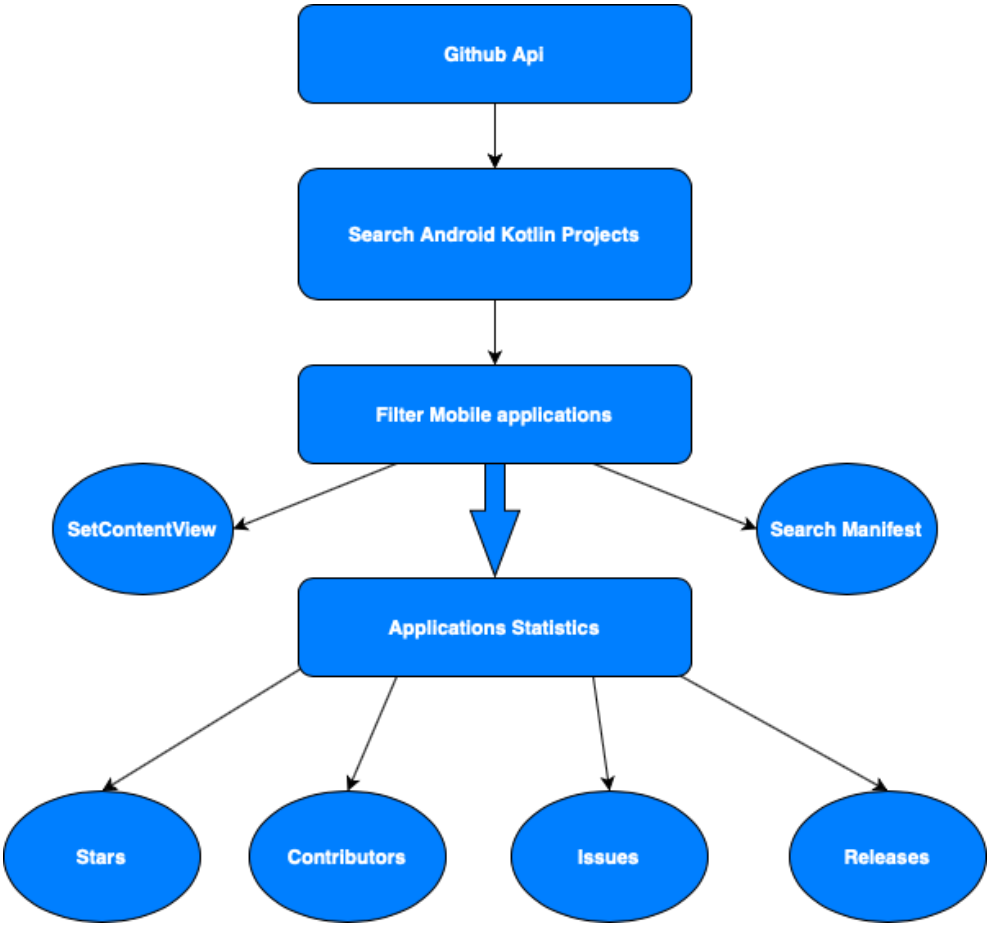

Figure 2.2: Data mining procedure

In Figure 2.2, we see the steps of our mining procedure, starting from the GitHub API, all Kotlin Android repositories were collected then filtered according to Android applications characteristics, where every application should include at least one manifest file and one call to the function setContentView(). After filtering the applications and excluding repositories that are out of our interest, we form a list of all applications and then started to collect the information about each application in the list.

2.1 Search Kotlin Android Projects

The GitHub API limits searches to 1000 results, this slows the speed of data mining. To overcome this issue we divided the requests according to a data range. A list of data range values was provided inside an input text file passed to the program we are

using to request the API. Looking into all repositories created before January 2020 using Android and Kotlin as parameters with the API request. Choosing October 2017 as a reference for the last update, which is the release date of Android Studio 3.0, when Kotlin was included as an alternative to the standard Java compiler. Storing all the application updated after the threshold date and discarding the repositories that are not updated. Having a big set of repositories that will be used as input while filtering mobile applications and discarding applications tools. In the next section we will see the inclusion and exclusion criteria that we followed.

The bellow script is used to look for repositories and then comparing the last update with the threshold date before saving repository name:

```
1    url_all= String.format(baseUrl,dateRange,page,consumerKey,
     consumerSecret);
2    JSONObject json_page_object=Json_Read_Tools.readJsonObjectFromUrl(
     url_all);
3     //items Contains a JSON array of repositories details
4    JSONArray json_page_array= json_page_object.getJSONArray("items");
5    for(int i=0;i<json_page_array.length();i++) {
6       JSONObject myobject=(JSONObject) json_page_array.get(i);
7       String repos_name=myobject.getString("full_name");
8       //Extract the YEAR and MONTH of updated_at string
9       String update_date=myobject.getString("updated_at");
10      Date date2=format.parse(update_date);
11
12      if (date1.compareTo(date2) <= 0) {
13        num2017++;
14        if(repos_name!=null) {
15            bw.write(repos_name); //Write using BufferWriter
16            bw.newLine();
17            bw.flush();
18        }
19    }
```

In the first step, repository Id with the account and project name that are updated after October 2017 are set inside the table with repository Id as unique Id so we avoided duplication of repositories that could happen because of forking a repository.

2.2 Inclusion and Exclusion Criteria

As a result the data mining of repositories using Kotlin as a language, a list of more than 46,000 repositories was obtained. And for the fact that our study is based only on mobile application, a set of filters is applied on the output list of repositories collected.

The list of repositories obtained in the previous part included libraries,utilities and applications. In order to differentiate between these repositories and exclude all repositories that are out of our interest , we specified some characteristics that should be presented in every Android Application and we started searching inside the content of each repository using Java classes that fetch the API of GitHub and exclude the repositories that don't have the minimum characteristics of an Android application discussed in this section.

An example script shows the procedure for looking into repository files content:

```
1    String url_manifest=String.format(url, repository_uri,consumerKey
     ,consumerSecret);
2    JSONObject json = Json_Read_Tools.readJsonObjectFromUrl(url);
3
4    int count=json.getInt("total_count");
5    if(count>=1) {
6        bw.write(repository_uri);
7        bw.newLine();
8        bw.flush();
9    }
```

2.2.1 AndroidManifest.xml

Manifest file is the key file for every Android application, the "AndroidManifest.xml" contains all essential information required by the operating system concerning the application,it works as a bridge between Android developers and the Android platform.[**Android:manifest**]

Since it's mandatory for each Android mobile application to have a manifest file,we developed a Java class that search for the number of "AndroidManifest.xml" for each repository inside the list of all repositories. As a result we saved the list of filtered repositories that contain at least one manifest file, while discarding the applications that returned empty result from the GitHub API. The number of manifest files was obtained by searching in the content of each repository for a file name "AndroidManifest.xml" using the following url:

`api.github.com/search/code?q=manifest+filename:AndroidManifest.xml+repo:<repository-name>`

Because of the big number of projects to filter and the issue of rate limiting of using GitHub api we were able to do only 60 request per minute,so a 2 seconds sleep was the solution of this issue while the only disadvantage was that it took around 26 hours to search into the contents of all repositories.

$$Time_{minutes} = Number_{projects}/Request_{/minute} \tag{2.1}$$

After fetching the content of 46,558 repositories, we found that 44,916(96%) repositories contain at least one manifest file, so that around 4% of the repositories were deleted from the database and excluded in the future analysis.

2.2.2 SetContentView

SetContentView() is one of the main functions for developers working with Android applications. it's used to display the layout created through XML on the screen. It is a part of Android.app.Activity class, and it's used when an activity starts inside the OnCreate() function and it need to know what user interface XML to display to user. This XML file is passed as a parameter. As a second step of filtering repositories, deleting all the repositories that are out of our interest, we fetched the content of repositories that contains at least one manifest, which was the output of the previous filtering criteria. So that after this step we assumed that the new list of repositories represents mobile applications.

As a conclusion we used the function setContentView() to complete our inclusion and exclusion criteria to include only applications that had an actual GUI, and as a result we saved the repositories that contains at least one call of the function SetContentView(). The following url is used to search the number of SetContentView inside a specific repository passing repository name as parameter:

```
api.github.com/search/code?q=setContentView+repo:<repository-name>
```

We found that after this step, that out of 44,916 repository that contains at least one manifest file, there's 42,931 application with more than one call to the function "setContentView()" (around 95% of files containing manifest file and 92% of projects updated after October 2017).

2.3 Statistics Search

After getting the repositories that represents Android application on GitHub, we started fetching statistics for each repository. All the information needed about repositories are saved into a database2.1.

2.3.1 Repositories Releases

The output list of Android applications was divided into different sets where each set has different characteristics related to the number and date of releases.

The example script bellow was used to fetch the number of releases and information about each release from GitHub the set of Android applications we have as an output after the inclusion and exclusion filters.

```
1     JSONObject myobject=(JSONObject) json.get(i);
2     String[] published_at=myobject.get("published_at").toString().
      split("-");
3     String year= published_at[0];
4     String month= published_at[1];
5     int date = Integer.parseInt(year+month);
6     String node_id = myobject.getString("node_id");
7     String tag = myobject.getString("tag_name");
8     Dbhelper.insertRelease(node_id, owner, repo, tag, date);
```

All the information was fetched from the GitHub API using the url bellow:

`https://api.github.com/repos/<repository-name>/releases`

A part of our analysis is based on the comparison of different releases of Android open source mobile applications. Starting from this definition we applied a set of filters to divide repositories. The number of releases with the tag of each release of the applications is saved into Sqlite database, with the date of publication of each release. On the other hand, all projects with zero or one tagged releases are used to compute the diffusion matrices explained in the third chapter using the master branch as a point of reference.

We found that out of 42,931 projects that represents mobile applications, only 3,894 applications had at least one tagged release(9% of total number of Kotlin Android open source applications on GitHub), while 39,037 projects are with zero number of releases(91%).

Applications Set	**Count**
Total number of Android applications	42,931
Number of application with at least one release	3,894
Total number of tagged releases	∼27,610
Tagged releases before October 2017	∼6760
Tagged releases after October 2017	∼20850

Table 2.1: Distribution of releases over applications

2.3.2 Issues

GitHub uses a tracker called issues to keep track of tasks, improvements, and bugs for projects. Issues for open source projects are shared and discussed in public where developers can collect user feedback and report software bugs. Issues make it easier to report software bugs but also to include more in-depth documentation. As part of the code analysis we collected from GitHub API for each repository, the number of

issues and then we collected for each month the sum of issues. All the data collected are saved into a database table for future use. The Information was collected using the url: `https://api.github.com/repos/<repository-name>/Issues`

The Issues API contains information about project owner, the assignees of issue, date of creation, last update and if the issue is closed with the date of closure. besides that information we can find links to other APIs that contains more details about the issue and the users collaborating with the issue.

2.3.3 Stars

Stars on GitHub can be used by logged-in users for different reasons. A user can follow a project updates by adding it to the list of starred repositories, other people use stars to indicate that they liked the project. GitHub stars can also be used to measure the popularity of applications,with the assumption that repositories with more stars might be known to many people and projects with only a few stars may be relatively unknown. However, we faced some limitation while fetching stars of repositories from GitHub API. GitHub did not mention inside the API the date when the user added the star , for this reason we collected the number of stars for each repository and then joining number of stars with the last release for each application.

2.3.4 Contributors

Contributors on GitHub are developers from outside the core development team of the project that contribute some changes to a project, it's one of the GitHub features that help sharing and collaboration in developing a project.

GitHub provides an API that represents the contributors of each open source applications, so using this API we were able to collect the number of contributors, saving the value in the database table of projects information. The number of contributors can be one of the vectors to calculate popularity of an application by referring to the number of developers that are interested in the contribution in Kotlin mobile applications . Information about contributors can be fetched using the url:

`api.github.com/repos/<repository-name>/contributors`

However requesting the above API returns as a response the list of contributors that contributed in the project with a set of references to information related to the contribution. Our interest in this study is to collect the number of contributors that was calculated by getting the size of the response array where each object represents different user.

2.4 Results and Observations

Our main focus is to study the transition of Android applications from Java to Kotlin, so we started fetching for projects that have "Kotlin" as a language on GitHub Android repositories. We included only Android applications that are updated by adding the language Kotlin or applications created recently and using Kotlin.

For this reason we can see in the results provided in this section that the numbers increased in a rapid way with respect to the statistics of the same set of applications.

While fetching and filtering the data about repositories, we saved the number of projects that passed successfully the filters in a local text file and in figure 2.3, we see the number of projects included in our study after each step of filtering procedure.

After collecting all the needed information, we developed a Java class that downloads the list of Kotlin mobile applications that will be used to calculate the matrices containing more specific details about the existence of Kotlin programming patterns inside the applications, and to study the usage of the new Kotlin features. For applications that are without any tagged release, we downloaded the branch master, assuming that it represents the last release of the application. Repositories from GitHub were cloned using the Curl command bellow:

```
curl -L -o <file-name>.zip http://github.com/<current_repos>/zipball/
    master/
```

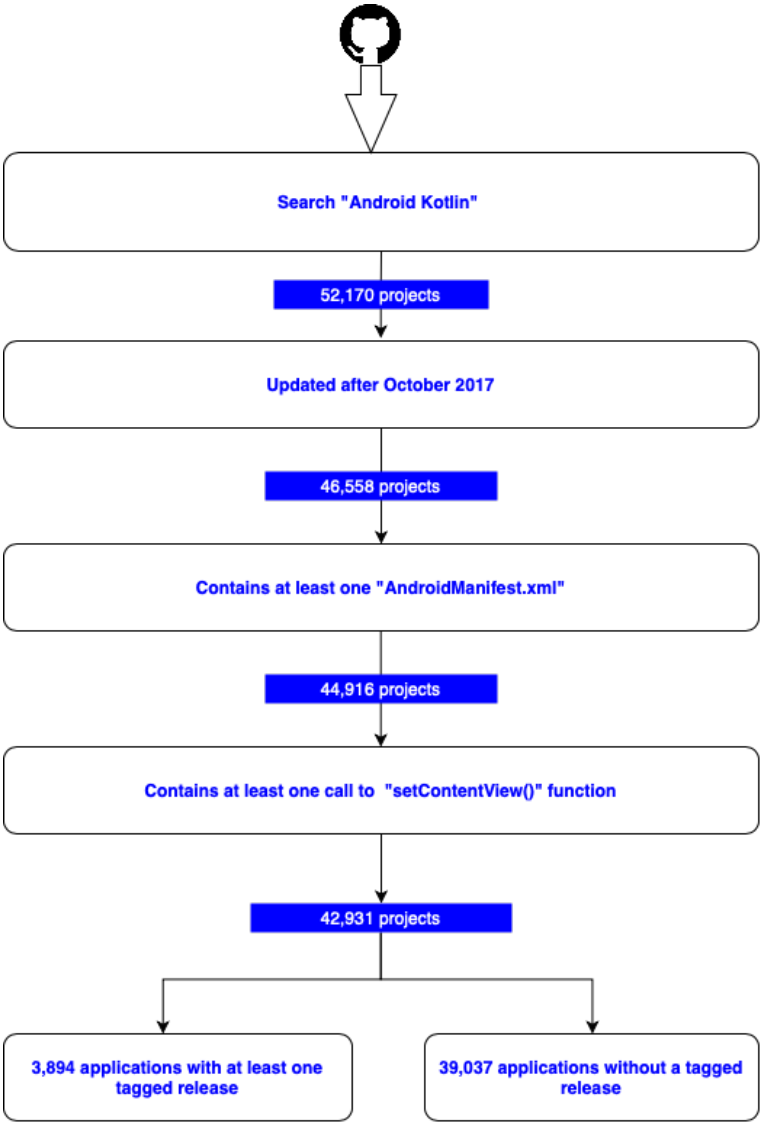

Figure 2.3: Number of GitHub repositories after each step of filtering

Applications Releases

In this section we defined two different sets of releases depending on release date. Figure 2.4 shows the number of releases before October 2017, which represents a smaller set of releases with respect to the set of releases after October 2017 :

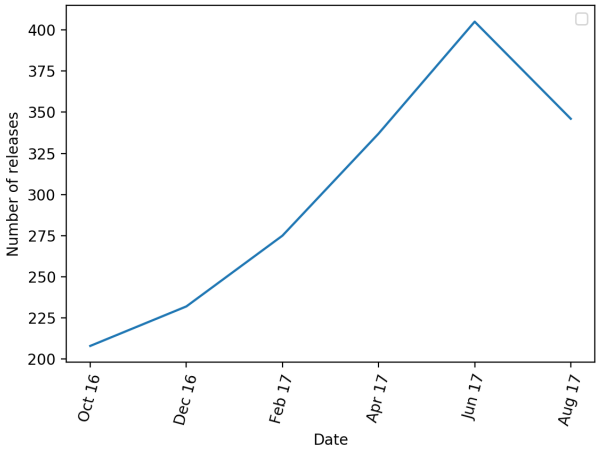

Figure 2.4: Number of Releases With Respect to date before the official announcement of Kotlin as alternative to the standard Java compiler

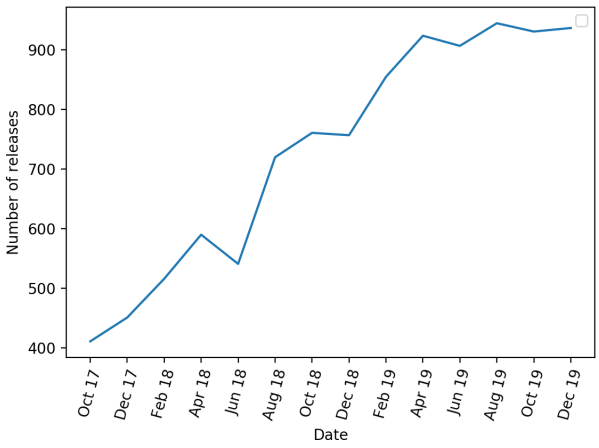

Figure 2.5: Number of Releases With Respect to date after the official announcement of Kotlin as alternative to the standard Java compiler

We can see from the two figures(2.4 and 2.5) that the number of tagged releases of Android mobile applications is increasing so fast with respect to the number of applications that are released in the same period. The number of tagged releases between October and December 2019 was around 1000 release per month, while in the same period in 2016, the average was around 200 release per month.

Applications Set	Num Apps	Average stars
ZERO Issues	35,875	3
Num Issues <= 10	3,546	63
10 < Num Issues <100	451	675
Num Issues >= 100	30	3147

Table 2.2: Applications sets according to number of issues

In table 2.2, we see four different sets of applications while to total number of stars for all applications is 713,171 and the total number of issues found on GitHub API was 29,296 we see that only 4,027 applications used issues as a way to track tasks and share bugs on GitHub interface. And as we can see that most applications has less than ten issues. on the other hand, the percentage of stars was calculated with respect to the number of applications in the set, We can notice that the increase in the number of issues reflects a remarkable increase in the average of stars of repositories. The average of stars on the set of applications with more than 100 issues(3,147) is 4 times greater than the average of stars on the set of repositories with issues between 10 and 100(675).

Applications Set	Num Apps	Average stars
ZERO Releases	36,027	7
Number Releases <= 10	3,173	69
Number Releases >10	561	450

Table 2.3: Number of stars with respect to number of releases

We divided the list of mobile applications into three different sets. One set with applications without any released tag, while for released applications we a set for applications with less than 10 releases and a another for all other projects with number of releases > 10. as we can see in table 2.3, we can see the number of applications on three sets with respect to the number of releases. Using the data collected from GitHub API we calculated the average number of stars with respect to the number of applications in each set as we can see in the second column.

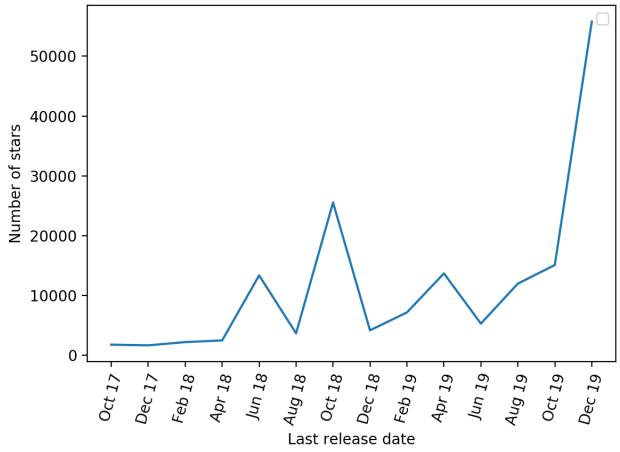

Figure 2.6: Sum of stars with respect to last release date

In figure 2.6, we divided applications according to the month of the last release, and then calculated the sum of starts for application released in the same month, as a way to show number of stars according to last release date.

The increase in the number of contributors of an application using 'Kotlin' language shows that the language is more known between developers. GitHub provided a tool to show the history of contributions in a project using a graph that shows the number of contributors to master branch excluding merge and empty commits. Graphs **??** and 2.8, show for a sample applications the contributions with respect to the date for some sample applications with high number of releases as for applications with high number of contributors.

"SubwayTooter"[20] is the application with highest number of tagged releases(399 releases) in the filtered list we have as output after the inclusion and exclusion criteria we followed. Graph 2.7 , we see the number of commits done by 25 contributors between April 2017 and February 2020.

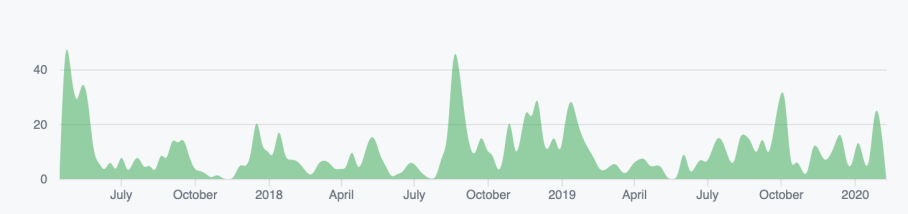

Figure 2.7: SubwayTooter Commits by contributors with respect to time

On the other hand,"susi_Android" is the application with the highest number of

contributors(142) in our filtered list with nine tagged releases. we see in graph 2.8, the change in the number of commits by all the project contributors with respect to time.

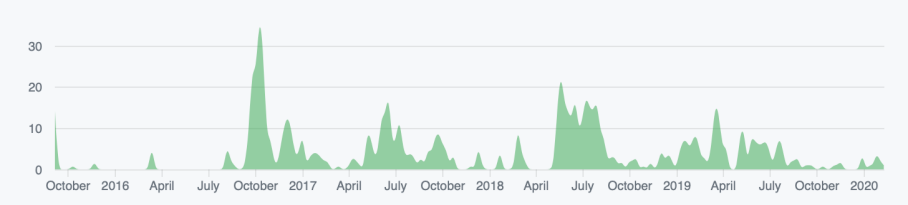

Figure 2.8: susi_Android Commits by contributors with respect to time

Git repository hosting service GitHub represents to developers, in an analogical manner, what social media platform Instagram does to its target audience,like photographers. A developer on GitHub follows issues searching for solution for bugs as for solving the bugs in other projects. Using the same sets of applications used in table 2.2, divided according to the number of issues, we calculated the percentage of contributors and the percentage with respect to the number of contributors on all projects.

Applications Set	Num Contributors	% Contributors
ZERO Issues	41,235	∼77.5%
Num Issues <= 10	6,396	∼12%
10 <Num Issues <100	4,147	∼7.8%
Num Issues >= 100	1,454	∼2.7%

Table 2.4: Contributors sets according to number of issues

As a result from the database data, we can see in table 2.4 that the increase in the percentage of contributors in mobile applications reflects a decrease in the number of issues and vice versa.

Chapter 3

Kotlin Development Practices

In this chapter, we are going to calculate for each project in the output list of Kotlin Android mobile applications of chapter 2, the percentage of Kotlin code and the usage of Kotlin development practices. Kotlin introduced some additional features so that fewer lines of code can have the same functionality of Java functions. Some of these features are data classes, nullability, mandatory casts and long argument lists. Using the line oriented search tool ripgrep [21], we developed a python class that recursively searches directories for a regular expression pattern. Parsing the list of all downloaded applications, we defined a list of regex, to search the existence of operators that represent the new features of Kotlin. We defined a set of rules to study the usage of the new featues by Kotlin developers. In figure 3.1, we can see the steps of parsing applications data using the tools (cloc and ripgrep), Starting from the list of downloaded applications from github.

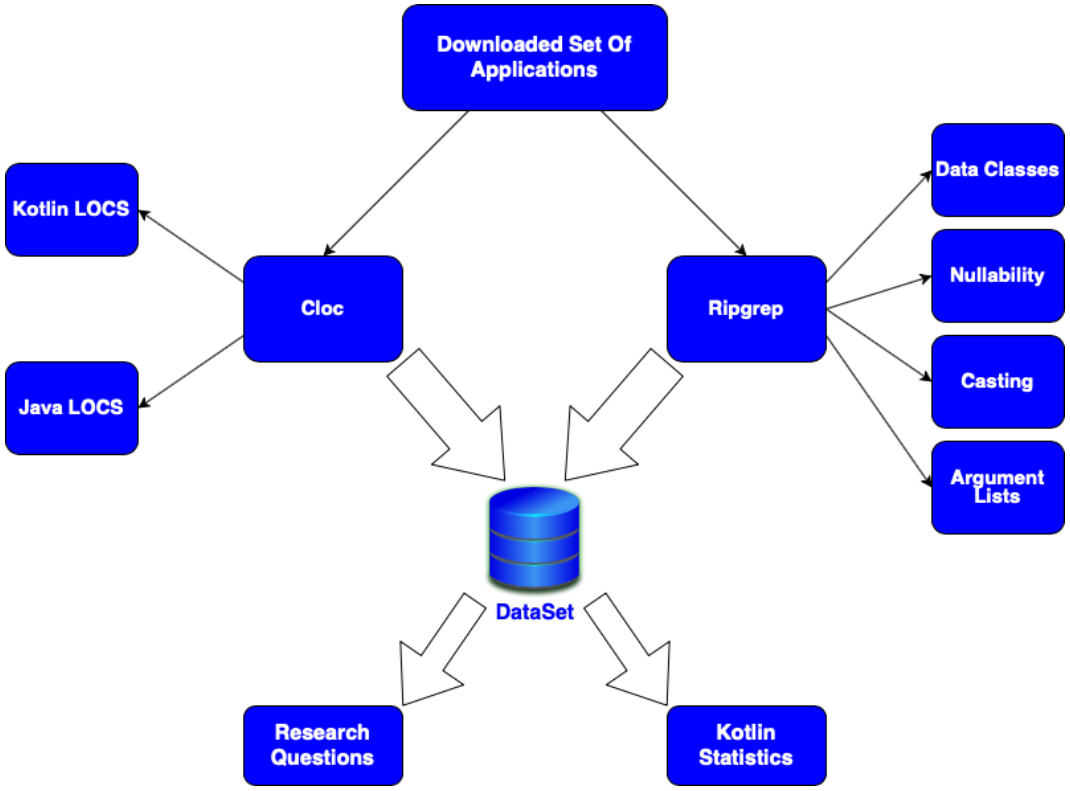

Figure 3.1: Procedure to Answer Research Questions

Using CLOC tool[22], we are going to divide applications into 4 different sets according to the percentage of Kotlin code. In figure 3.2, we can see the structure of the database table used to save collected data. In this table we assigned application name as Id so no Duplication of applications with same name may happen.

CLOCStatistics	
Application	TEXT
Issues	INTEGER
KotlinPercentage	INTEGER
KotlinLOCs	INTEGER
KotlinFiles	INTEGER
JavaLOCs	INTEGER
JavaFiles	INTEGER

Figure 3.2: Database table of Cloc Statistics

3.1 Data Classes

Data classes are containers of data, same as classes in Java with the reduction of a lot of boilerplate that Java forces developers to generate. Developers end up with a code that is easier to understand and to maintain. Developing a simple Java application may need some classes to be involved, where for each class developers must define class fields, then for each property we need getters and setters with some additional methods to be employed like 'equals' and 'hashCode'. Kotlin replaces classes by a more simple feature called 'Data class', where properties are declared in the constructor and the compiler is able to auto generate all the boilerplate code.

Java vs Kotlin classes

Bellow we can see the comparison between Java classes with all the boilerplate code and a simple Kotlin data class that have exactly the same functionality but with the reduction in lines of code.

```
1  public class Person {
2      private String name;
3
4      public String getName() {
5          return name;
6      }
7      public void setName(String name) {
8          this.name = name;
9      }
10
11     @Override public boolean equals(Object o) {...}
12
13     @Override public int hashCode() {...}
14
15     @Override public String toString() {
16         return "Person{" name= " + name  + "}";
17     }
18 }
```

In Kotlin all the declared functions in the example are generated automatically by the compiler, bellow we can see the a sample of a Kotlin data class, where in one line we can have the same functionality as a long Java class:

```
1 data class Person(var name: String)
```

The compiler in Kotlin automatically generate functions of the properties that are defined inside the constructor, while properties outside the constructor are excluded from the generated implementation. In the example bellow we can see a data class generation with as the above example adding a new parameter outside the constructor. So property 'age' is excluded from the automatic generated functions, while 'name' can be used inside the toString(), equals(), hashCode(), and copy() implementations.

```
1 data class Person(val name: String) {
2     var age: Int = 0
3 }
```

3.2 Nullability

Null values in Java programming context used to represent the absence of a result, and it is different from 0 or blank. The ability to accept null values is referred by Nullability. Null values may lead to NullPointerException in case of lack of null checks. Instead of checking null values for every value,Java 8 introduced a new class, "Optional" class is a container object used to represent null with absent value. This class added a new way to handle null values using 'available' or 'not available'.

Kotlin aimed to eliminate NPE's from our code. The type system recognizes nullable references for non-null references. Kotlin introduced a way to deal with null values, developers can decide to allow null values by declaring variables as we can see in table 3.1 [23].

Declaration	Notes
var b: String? = "abc" b = null val l = b.length	Variable 'b' can be null when "?" is used
var a: String = "abc" a = null	assigning null to variable 'a' leads to compilation error
val a = "Kotlin" println(a?.length)	Unnecessary safe call (a?.length) a cannot be null
val b: String? = null println(b?.length)	Safe calls is used when variable accepts null as a value. They are useful with chains
val l = b?.length ?: -1	Elvis Operator "?:" set default value if the variable is null value
val l = b!!.length	Operator (!!) converts any value to a non-null type and throws an exception if the value is null.
val a: Int? = a as? Int	safe casts "as?" : return null if the attempt was not successful

Table 3.1: Null Safety in Kotlin programming language

3.3 Mandatory Casts

Type safety is a method provided by programming languages to guarantee that a type mismatch between a variable and a value that is stored may not lead to an issue. To enforce type safety in Java, the class tag of the object must be controlled before every operation on the object. This will help make sure the object's class allows the operation. Valid values are verified at run-time before being stored in variables. Kotlin introduced new ways to perform type checks and casts, to ensure type safety of operations by involving generics at compile time. While at run time generic types hold no information about their actual type. Casting is necessary in programming world, Kotlin accepts the challenge by introducing nicer ways for type checks and to cast variables facing the issue of the wrong casts could be accepted by the compiler and result an exception in runtime as in Java.

'is' and '!is' Operators

A developer can check if an object accepts a given type at run-time by using the 'is' operator or its negated form '!is'. This operators can be used as smart cast where it automatically casts the object in runtime.

In the example bellow we can see that after checking if the variable x is a String using 'is' operator, the variable is automatically cast to string on the right side of the '&&' operator.

```
if (x is String && x.length > 0)
    print(x.length)
```

Cast operator

Different ways to cast operators are provided in Kotlin, they mentioned a "Safe" and "Unsafe" cast operators. Since the cast operator throws and exception if the cast is not possible, Kotlin defined the casting using 'as' operator as unsafe cast. In the casting example bellow, the code throws an exception if y is NULL.

```
val x: String = y as String
```

On the other hand, Kotlin programming language provided safe casting using 'as?' that returns null in case of casting failure. Using this feature, programmers can avoid casting exceptions. In the example bellow we see the usage of the safe cast operator, However if y has a non null type, the result of the cast is nullable.

```
val x: String? = y as? String
```

3.4 Argument Lists

One of the challenges that face every programming is dealing with functions that should have a variable number of parameters, some cases in Android applications, for example the execute(Params... params) function from AsyncTask or the function printf(String format, Obj... args). Kotlin provides defining a parameter of a function as vararg. Vararg can take n number of parameters where the value of n can be decided at run-time.

```
fun format(format: String, vararg args: Any)
```

However, Kotlin adopts different solutions to deal with functions with long argument list, these features make the code more readable and flexible. First, Default arguments, so that function parameters can have default values to avoid null or empty parameters value during run-time. In the function bellow the value of the Int 'bar' is set by default to 0, on the other hand, no default value for the variable 'baz'.[24]

```
fun foo(bar: Int = 0, baz: Int) { /*...*/ }
```

Named arguments is another way to deal with long argument lists, so that function parameters can be named when calling functions. This is very convenient when a function has a high number of parameters or default ones.[24]

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {/*...*/}
```

In the above function we can see a function with named parameters provided with default value, while the only parameter without a default value is str.

3.5 Code Inspection

In this section we are going to inspect the list of downloaded projects, searching for code patterns related to Kotlin programming language and saving all the information in the database table 3.3.

CodePatterns	
Application	TEXT
DataClass	INTEGER
SafeCalls	INTEGER
ArgumentLists	INTEGER
SafeCast	INTEGER
UnSafeCast	INTEGER
DefaultValues	INTEGER
NullAssertionOp	INTEGER

Figure 3.3: Database table of Ripgrep Statistics

3.5.1 Inspection Tools

To Inspect the list of projects we developed a python script that path into list of directories, then using the following tools we were able to calculate the data needed in our study.

Regular Expressions

A regular expression is a sequence of characters that define a search pattern. We used regex to inspect Kotlin programming patterns that were introduced in this chapter. We defined for each of the practices a Regular Expression according to the programming syntax.

Practices	Regular Expression
Data Classes	data\sclass
Safe Calls	\?\.
Safe Casting	\sas\?\s
Unsafe Casting	\sas\s
Default Values (accepts null)	.+\s?:\s?.+\?\s=\s?.+
Operator(!!)	!!
Argument Lists	vararg

Table 3.2: Regular expressions represent the Kotlin practices patterns

In table 3.2, for each practice we defined the syntax pattern of the operators, for mandatory casts we will search the 'safe'(as?) and 'unsafe'(as) casting as defined by Kotlin documentation. In Kotlin variable initialization, developer can decide if the variable can accept null value or not. We used the pattern(<declaration> : [Type]? = <value>) , to search the number of declarations of variables that accepts null. The not-null assertion operator (!!) converts any value to a non-null type and throws an exception if the value is null. Functions with Kotlin can accept variable number of parameters, using the new feature 'vararg', where the number of parameters is decided at run-time. Data classes is one of the most important practices with the advantages of reducing the number of lines and complexity of the code. We search the occurrences of all patterns inside all the downloaded applications to see how frequently developers are using the new practices of Kotlin.

Count Lines of code

'CLOC'[22] is a tool used to count blank lines, comment lines, and physical lines of source code inside a directory tree of a project in many programming languages.

Using cloc, starting from the list of downlaoded application, we calculated the number of Kotlin code and files as for Java files and line of code with the average of Kotlin code. Bellow is the of cloc with a 'JSON' formatted output :

```
> cloc <dir-name> --json --out="<output-file>.json"
> Json output sample:
    "<language>" :{
      "nFiles": number,
      "blank": number,
      "comment": number,
      "code": number},
```

ripgrep

'ripgrep'[21] is a line-oriented search tool used to search for regular expression pattern inside a directory, and all hidden files and files listed in .gitignore are skipped. We used rigrep to search the regular expressions of Kotlin operators and study the usage of these operators with the mobile applications.

3.6 Research Questions and Metrics

In this section, we will define the three Research Questions to pursue the goal of our study. After inspecting and collecting all the information about lines of code of each of the mobile applications in our data set, we calculated the metrics of the research questions:

3.6.1 Diffusion

Diffusion is used to calculate the adoption of Kotlin on Android applications on GitHub. For all the Kotlin applications in our data set, we supposed that master branch is the last release of the application to make the study on a stable branch. To calculate the diffusion of Kotlin applications we defined the metrics bellow:

Kotlin Relative Lines of code

KRL is the number of Kotlin lines code over the total amount of production LOCs of the project(*.Java + *.kt).

$$KRL = Count_{KotlinLOCs}/TotalNumberOfLOCs_{Kotlin+Java} \tag{3.1}$$

Kotlin Relative Files

KRF is the average of Kotlin files in all mobile applications with respect to the total number of production code files.

$$KRF = Count_{KotlinFiles}/TotalNumberOfFiles_{Kotlin+Java} \tag{3.2}$$

Kotlin Featuring Projects Ratio

KFPR is the ratio of production projects of a set featuring at least a Kotlin file. In our list of applications, we fetched only Kotlin Android applications updated after October 2017, so we expected that almost all applications contains at least one Kotlin file.

$$KFPR = ProjectsCount_{KotlinFiles>=1}/TotalNumberOfApplications \tag{3.3}$$

Kotlin-Majority Projects Ratio

KMPR calculated the ratio of applications with majority of Kotlin LOCs over the total count of applications. The Kotlin-Majority Projects are the projects with number of Kotlin LOCs greater than number of Java LOCs.

3.6.2 Evolution

To study the evolution of mobile applications, we defined a list of metrics that depends on the different releases of the applications. This section of our study will use only applications with more than two releases to calculate the statistics, so that only 1,793 applications out of 33,267 will be involved. We calculated for different releases of each application the number of LOCs and files of both Kotlin and Java programming languages.

Kotlin Adoption Relative Releases

KAR is the ratio of tagged releases that featured less than 50% of Kotlin code.

$$NumberofReleases = Releases_{KotlinCode/TotalNumberOfproductioncode} < 0.5 \tag{3.4}$$

$$KAR = NumberOfReleases_{3.4}/TotalNumberOfReleases \tag{3.5}$$

first we calculate the ratio of usage of Kotlin code over the total number of production code as in diffusion metric KRL. then using the number of releases3.4 over the total number of releases we can obtain the KAR.

Kotlin Majority Relative Releases

KMR is the number of tagged releases with majority of Kotlin code over total number of production code lines (Java + Kotlin). First using the percentage of Kotlin code, we selected releases with Kotlin percentage greater than 50%.

$$NumberofReleases = Releases_{KotlinCode/TotalNumberOfproductioncode} > 0.5 \quad (3.6)$$

$$KMR = NumberOfReleases_{??}/TotalNumberOfReleases \quad (3.7)$$

Only Kotlin Relative Releases

KOR are the releases with 100% of Kotlin code, without any line of code. For application releases created after October 2017 with an initial language as Kotlin are expected to be Java free. To calculate the result of KOR, we searched the number of pure Kotlin applications then divided the result over the total number of releases.

$$KOR = NumberOfReleases_{\%Kotlin=100}/TotalNumberOfReleases \quad (3.8)$$

3.6.3 Popularity

Popularity of applications is calculated by number of users, while popularity of programming language in our study is measured using the relation between the collected information about repositories and the percentage of Kotlin code in Android applications using Kotlin at least once. To solve the third research question if development with Kotlin have an influence on the success of released applications we sought a combination between results of mining and parsing repositories.

3.7 Related Work

With the revolution of Android mobile applications, many studies were done to study and characterize the transition to Kotlin of Android Apps.

Coppola et al.[**article:KotlinToJava**], Analyzed a set of open-source Android applications on F-Droid, Play Store, and GitHub. The objective of the study was to evaluate the transition of this applications to Kotlin throughout their lifespan. They started by mining projects from F-Droid and connecting the results with Google Play Store. On a set of 1,232 projects, researchers collaborating in the cited study, defined a set of eight metrics based on the diffusion, evolution and popularity of applications. Coppola stated that near 20% of applications adopted Kotlin, and 12% with a majority of Kotlin over Java code. Most applications adopted Kotlin,

had a quick transition to Kotlin and projects featuring the new language had a higher average of popularity metrics according to the occurrence of Kotlin and number of stars on GitHub repositories.

Kotlin seems able to guarantee a seamless migration from Java for Android developers. Coppola et al.[**article:KotlinToJava**] concluded with "the adoption of the Kotlin language is rapid (when compared to the average lifespan of an Android project) and seems to come at no cost in terms of popularity among the users and other developers"

3.8 Results

In this section we answer the research questions of this chapter, with the combination with the results of data mining.

3.8.1 RQ1: Diffusion

To calculate the diffusion metrics, we measured the percentage of Kotlin for all downloaded applications(33,267), taking into consideration that the list of empty repositories and duplicated applications was excluded. As a result we were able to do the study on around 33,267 application according to the version of code base collected in the end of December 2019. Starting from the data in the database table in Figure 3.2, we were able using a set of queries to calculate the diffusion metric. In Table **??** we can see an example of some applications with Kotlin Relative lines of code and Kotlin Relative Files, In addition to the number of issues,contributors and stars of the application.

Application	KRL	KRF	Issues	Contributors	Stars
gobAndroid	0.88	0.79	62	9	171
and-bible	0.78	0.41	136	10	210
DroidPersianCalendar	0.69	0.74	50	25	376
Put.io	0.76	0.66	15	10	133
libaums	0.5	0.69	24	10	592
Heimdall.droid	0.46	0.52	2	9	242
plaid	0.62	0.8	72	63	14953
Tusky	0.53	0.72	154	111	814
OpenLinkWith	0.55	0.68	10	2	127
SurvivalManual	1	1	35	4	436

Table 3.3: KRL and KRF on Sample Set of Application

Kotlin-featuring Projects Ratio

For the list of projects mined from github, it was expected that most of projects feature at least one Kotlin file, since the filter of Kotlin language was used during the study.

We find that 33,212 out of 33,267 applications(99.8%) of applications in our data set feature at least on Kotlin file in the master branch on GitHub. While the existence of this applications that without any Kotlin file while fetching data from GitHub could be either because Kotlin was used in previous releases or it will be integrated in future release.

Kotlin-Majority Projects Ratio

To measure the ratio of applications with the majority of Kotlin with respect to production files, we used as input the result of KRF, then selecting only the applications with KRF>=0.5, so that the percentage of Kotlin files inside the application is greater than 50%. On the same set of applications mined from github, using a query we found that 32,877(98.83%) applications out of 33,267 are featuring majority of Kotlin files with respect to production files (Java + Kotlin).

3.8.2 Evolution

To measure the evolution of Android releases, and because of the time needed to calculate LOCs of all releases, we used a sample of 488 applications with a tagged release before October 2017 as a reference that the initial language of the project was Java.

Kotlin Adoption relative releases: We found that 19/458(4%) tagged release featured less than 50% of Kotlin code including releases with only Java. This means that these applications changed a small part of the base code from Java to Kotlin during the period between October 2017 and release date.

Kotlin Majority relative releases: To calculate KMR, we searched the releases with a majority of Kotlin code with respect to the production code, we got as an output that 207 releases used a majority of Kotlin with the existence of some Java LOCs. Based on the result over the total number of release included in the study we found that KMR = 252/458 (55%). Which means more than half the applications are still using Java in applications after more than two years of the official release of Kotlin.

Only Kotlin Relative releases Converting all source files from Java to Kotlin is a tool supported by google and can be used on Android studio. This tool may lead to some bugs, and must be checked after conversation. Many Android applications converts the base code from Java to Kotlin with the removal of all Java lines of code from the production code of the applications. We searched the releases with

100% of Kotlin lines code, and we found that 186 releases are featuring only Kotlin. So that KOR = 186/458 (41%).

3.8.3 Popularity

To measure the popularity of applications, we defined three parameters that indicates the usage and followers of a GitHub open-source application. First, The increase in the number of contributors of an application indicates that developers are contributing more with the applications featuring Kotlin which is a sign of popularity between programmers. Second parameter is the number of stars, which is the number of users that liked the repository on github platform. The third parameter is the number of issues, that is an important feature of popularity of an application, so that the increase in the number of issues indicates that the application developers team is tracking tasks, improvements and bugs continuously.

Kotlin Perctage	0-25%	25-50%	50-75%	75-100%	Total
Number of Applications	34	121	1,820	31,292	33,267
AVG Contributors	1.7	1.6	1.9	1.3	1.6
AVG Issues	34.8	0.8	2	0.6	9.5
AVG Stars	36.7	30	61.3	15.8	36

Table 3.4: Distribution of Contributors,Issues and Stars with respect to percentage of Kotlin

In table 3.4, the average of Contributors,Issues and Stars is calculated with respect to the number of applications of the set, while the last column represents the average number of applications,Contributors,Issues and Stars that is used as a threshold to measure popularity. The percentage was calculated using the following formula.

$$AVG(Contributors) = SUM(Contributors)/NumberOfApplications \quad (3.9)$$

$$AVG(Issues) = SUM(Issues)/NumberOfApplications \quad (3.10)$$

$$AVG(Stars) = SUM(Stars)/NumberOfApplications \quad (3.11)$$

we concluded that for applications with percentage of Kotlin greater than 75% , the parameters of popularity we used in this section are under the threshold, due to the high number of applications, where around 95% of application in our list belong to this set of applications. On the other hand, surprisingly we found that for applications with percentage of Kotlin below 25% have the average of Contributors,Issues and Stars is so high with respect to the threshold value.

3.8.4 Code Inspection

In this section we solve the main question of the chapter about the usage of Kotlin development practices in Android applications that are featuring at least one Kotlin file. To solve this question we used ripgrep [21] tool in a python script 3.8.4, that was able to fetch Kotlin files of all the applications searching for the regular expressions defined in table 3.2. All the data collected was saved in a database table 3.3. Then Querying the dataset we were able to calculate the statistics with respect to the percentage of Kotlin used in applications.

```
1     cmd = "rg --count \'" + str(regex) + "\' -z  " + os.path.join(
      apps_path, d)
2     proc = await asyncio.create_subprocess_shell(
3         cmd,
4         stdout=asyncio.subprocess.PIPE,
5         stderr=asyncio.subprocess.PIPE)
6
7     stdout, stderr = await proc.communicate()
```

We calculated the average usage of Kotlin practices over the full set of applications. We found that 17,029 out of 33,267 applications(51%) are not using data classes, while for the other 16238 applications, measuring the sum of data classes over the number of applications with at least one declaration of the "data class" resulted a 6.8 data class per application.

While fetching for mandatory casting, we fetched projects to find safe and unsafe casts, to see how often developers are using safe casting. We found that Android Kotlin developers are not using the recommended safe casting as expected which returns null in case of failure, 29,693 of the applications (89%) are not using safe casting (as?) which is a very high percentage with respect to the importance of the feature. On the other hand, unsafe casts "as" were used in 23,617 (71%) so that in case of failure , the cast operator throws an exception.

As Defined in section 3.5.1, (!!) operator returns NPE in case the value to be converted has a null value, We fetched all the applications source code searching for the (!!) operator. We found that 14457 applications (43.5%) out of the all the set of applications are not using Null assertion operator.

Kotlin solved to long argument lists issues by introducing "vararg" to deal with functions with variable number of arguments. The usage of vararg was not as expected taking into consideration the importance of this feature. We Found that 85% of the applications (28,302) are not integrating "vararg" into applications.

For the feature of nullability we considered two types of code practice. First, Variables with default values when declared with the "?" operator after the type

of the variable, means that the variable accepts null as a value, we fetched using the regular expression that represents the declaration of a variable with default value(e.g. var b:String? ="abc"). 18,206 (54.7%) of the applications initialized variables using the format given in the first row in table 3.1.

Second type of declaration is the safe calls (e.g. b?.length), which is useful with chains when some variables accepts null as a value. On the other hand, safe calls might be unnecessary for variables that cannot be null. The usage of safe calls in applications was interesting, where 70.4% of the applications used safe calls at least once in applications.

To finish the part of the study on the full set of application, we searched the applications that are not using any of the Kotlin features in the source code. We found that 5,022 (15%) applications are missing all the features discussed in this chapter, which is an interesting percentage on a set of applications that is using Kotlin as a main language for Android mobile applications development. In figure 3.4, we can see the distribution of the usage of Kotlin features on the full set of applications. Where the y-axis represents the number of applications using the practice over the total number of applications.

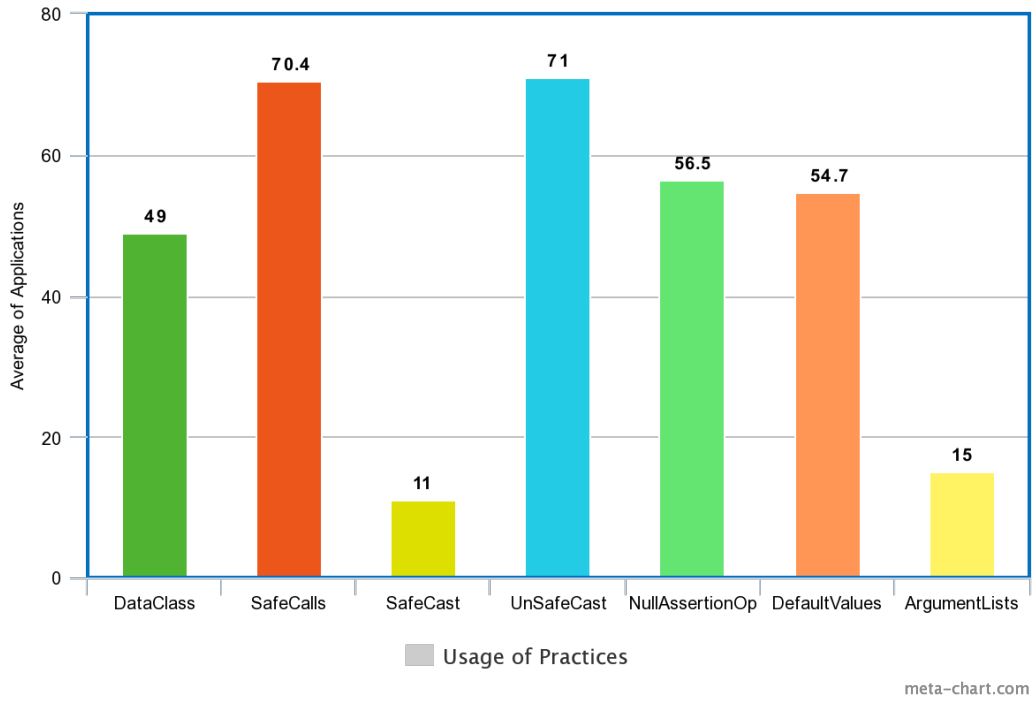

Figure 3.4: Distribution of Kotlin practices usage on the full set of applications

We concluded that practices like vararg and safe casts are present in a small set of applications, where practices like Unsafe casts are used in most applications. We

were expecting higher percentage of usage of the practices due to the advantages of each of them.

To measure the usage of the practices with respect to the percentage of Kotlin within the production code, using a list of queries to join the datasets of Lines of code and code pattern usage. We can see in the figure 3.5 **??** the results on four different sets of applications according to the percentage of Kotlin calculated in previous section.

In figure 3.5, On a set of 32 applications we can see the average of applications using Kotlin feature, 12 applications used data class at least one time in the base code of application, 18 application used safe calls,9 applications used vararg for a variable number of arguments of a function, 19 applications used Null assertion operator. Indeed, variables with default values were declared in 21 applications and Unsafe casts in 23 applications, while safe casting was found only in two of the applications in the same set. Both variables accepting null and unsafe casting may lead to a null result which may be as ignoring nullability feature provided by Kotlin. 8 out of 32 applications (25%) are not using any of the defined Kotlin practices.

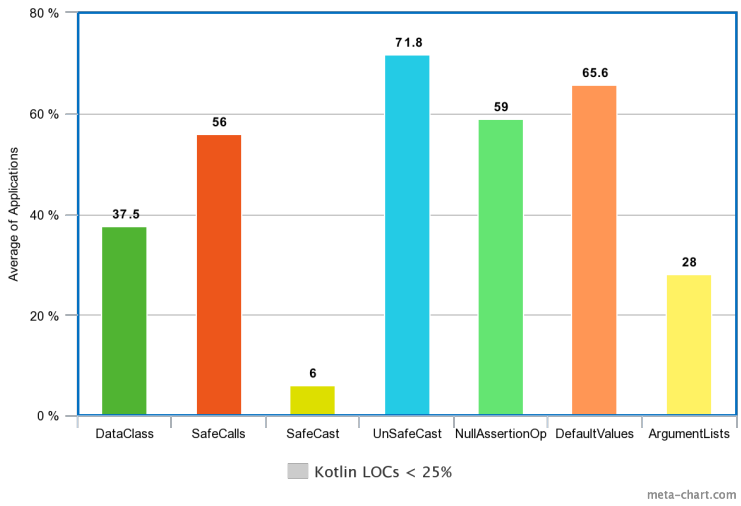

Figure 3.5: Usage of Kotlin practices on applications with %Kotlin < 25

In figure 3.6, Out of 112 applications, 47 applications used data class at least one time in the base code of application,22 applications used vararg, 65 applications used Null assertion operator which may return NPE, 90 applications used safe calls. Indeed, variables with default values were declared in 89 different applications and Unsafe casts in 82 applications, while safe casting was found only in 12 of the applications. 10.7% of the applications(12/112) are not using any of the defined Kotlin practices.

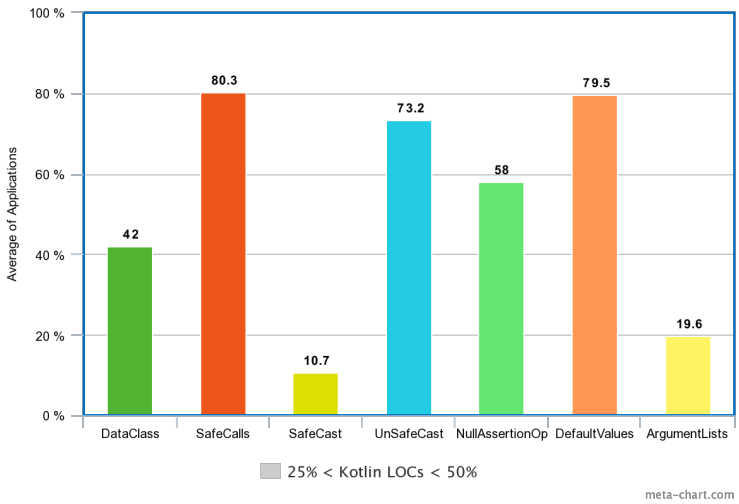

Figure 3.6: Usage of Kotlin practices on applications with Kotlin LOCs between 25 and 50%

In figure 3.7, Out of 1,448 applications, we found the usage of unsafe casting and variables accepting null in around 75% of the applications in this set. While the usage of safe calls was found in 1050 applications, and Null assertion operator in 974 applications. On the other hand we can still see than less than 50% of the applications are using the other practices. Data Class(622), Argument lists (333) and safe cast(210).

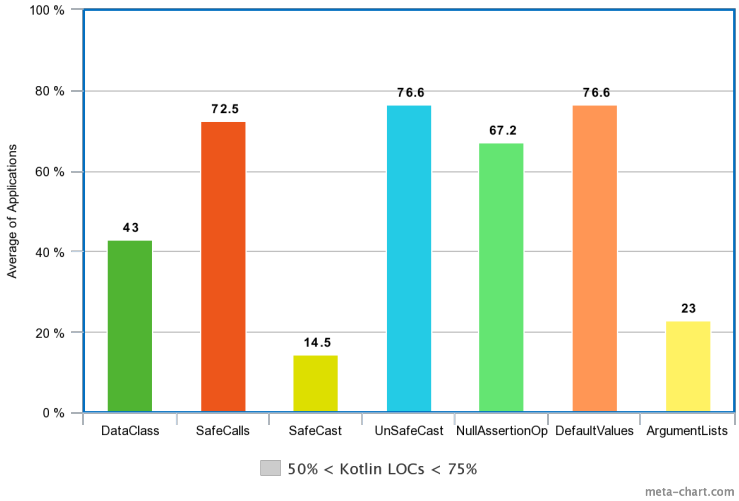

Figure 3.7: Usage of Kotlin practices on applications with Kotlin LOCs between 50 and 75%

The last set of applications is presented in Figure 3.8. It the set with the highest number of applications, 24,701 out of 30,852 has 100% of Kotlin code in the last release of the application. 15,017 used data classes, 21529 used safe calls and 17091 of the applications used null assertion operator at least once inside the application. However, Variables accepting null and unsafe casting were found in 20,265 and 21,695 respectively. Safe casts(3186) and argument lists(4358) having the lowest percentage of usage as in all the other sets.

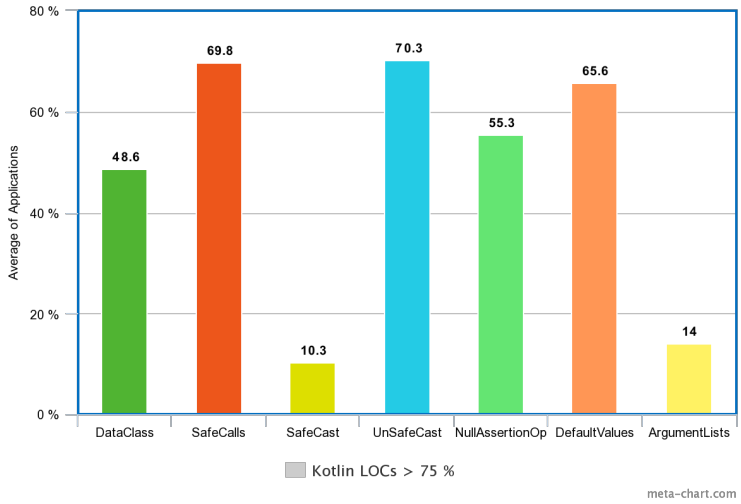

Figure 3.8: Usage of Kotlin practices on applications with more than 75% of Kotlin LOCs

Nullability was introduced to reduce Null Pointer Exceptions while the null assertion operator throws an exception if the value is null. The usage of (!!) operator occurred in more than half of the applications, which is a high percentage for an operator that might throw an exception. Safe casts were provided to reduce exceptions when variables have a null value. From the results we concluded that developers are declaring variables that accepts null values without using safe casting (as?) to reduce the exceptions. As the higher percentage of casting was done using the unsafe method(as).

Chapter 4

Code Quality analysis

4.1 Code Quality Overview

The Quality of a software is important for development teams as for individual developers. It reduces the cost of production by saving time and resources. Improving the quality of applications, makes it easier to collaborate between developers, fix bugs, re-use same functions in different applications so a better productivity is achieved. No single definition of code quality exists. It is all subjective, so that different developers and tools may have different definitions related to good/bad code based on the context. I.e. High quality code for a front end developer may be different with back end developer perspective.

For open source applications, quality of the code may be analyzed depending on code complexity. High quality code should be easy to read and modify by different contributors. However, every individual of the development team is responsible of the software quality. Improving the quality of applications can be done by following a consistent style, with a well documentation. According to a Richard Bellairs[25] ,the key aspects to follow for a higher quality code are:

Reliability

Reliable code means something is dependable and that it will give the same outcome every time.

Reliability testing checks whether the software can perform without failure over a specific operation for a specified period of time,so that software availability is calculated using the average time between failures.

To measure the probability of failure the formula bellow is used:

$$Probability = NumberOfFailingCases/TotalNumberOfCases \tag{4.1}$$

Maintainability

Maintainability measures how easily software can be maintained, it depends on the readability and consistency of the code base. It is inversely proportional to the amount of time to upgrade a code and the risk of errors after the update. However, maintainability is proportional to the testability and understandability of applications.

Mathematically, to measure maintainability of a code the following formula is used

$$Maintainability = TIME_{Implementation}/RISK \tag{4.2}$$

Testability

Testability in code quality analysis depends on the effort and effectiveness of the software. It is fully depending on numerous factors, it relies on how developers can control, observe, isolate and automate testing among various list of factors such as.

- Test cases.
- Software requirements
- Software properties (size, complexity ..)

Portability

Portability is the main key for development cost reduction, so that the same software can be used with different environments. There is no specific measure of portability. However, to ensure that a software is portable, developers have to regularly test the code on different platforms, rather than waiting software release. Using high compiler warning level with the enforcement of coding standard may help to have a portable software.

Reusability

Reusability is the usage of the same code to build new software applications. It is measured by the number of interdependencies, that can be identified using a static analyzer. In terms of development cost, code reusabulity can help developing teams to save time by using the same functions with some updates if needed. Reusing code often proves to be difficult, so that developers may prefer starting building software from scratch or only reuse a small fraction of existing code in new projects. This is because of code complexity and low quality software available.

4.2 Code Quality Metrics

To ensure the high quality of a software, developers might use various code quality metrics. In this section we discuss popular code quality metrics that can be used for the maintainability of the source code throughout the entire development life cycle.

4.2.1 Complexity metrics

The increase in the possibilities of paths of execution results a more complex program, so full tests, maintenance and understanding of software code becomes harder. So that the metrics defined in this section helps to estimate the complexity of applications

CC - McCabe's Cyclomatic Complexity

A software metric developed by T.J MaCabe[26] , used to measure the complexity of a program. It is based on independent paths in the code of a program. McCabe's CC can be measured in terms of functions, modules, methods or classes within a program. The code complexity is calculated by counting the number of execution paths based on the flow graph. Flow graph is composed of nodes and edges, so that a node represents a code block and edges represent the control flow between nodes.

Mathematically, the Cyclomatic complexity of the program can be defined using the formula where E,N and P are the number of edges, nodes and predicate nodes(nodes with conditions) respectively.

$$V(G) = E - N + 2 \tag{4.3}$$

$$V(G) = P + 1 \tag{4.4}$$

NOM - Number Of Methods

Number Of Methods metric is used to measure the complexity of a class by counting the number of methods in that class. The number of metrics is inversely proportional to the maintainability of a code, so that the higher the value of the metric, the lower the maintainability of the code. As sub metrics of NOM, we can discuss some special cases like: NPM - the number of public methods which is the count of public methods inside a given class so it can be used to measure the size of an API provided by a package. STAT - the count of statements inside a method.

NOC - Number Of Classes

Number of classes metrics measures the number immediate descendants of the class inside application package by computing the occurrences of classes.

WMC - Weighted Method Count

In 1994, Chidamber and Kemerer introduced the Weighted Method Count metric. that summation of the complexity of all class methods implemented in the analyze code. WMC metric depends on number of methods over the number of classes with the number of independent paths. To compute the average of weighted method count, the following formula is used:

$$WMC = (NOM/NOC) * \#paths \tag{4.5}$$

4.2.2 Dimensional Metrics

The purpose of dimensional metrics in code quality analysis is to measure the quality of software in terms of code sizes. Applications with large code base size may provide different features. On the other hand, the probability of complexity and bugs might be higher. dimensional metrics is based on the following metrics:

CHANGE - Number of lines changed in the class

CHANGE metric measures how many LOCs changes between two different versions of the same class of code. So it is not possible to compute this type of metrics on a single version of an application. CHANGE metric is usually used to analyze the evolution of applications. Classes with continuous modifications between versions may be a sign that the class is hardly maintainable. Change in a class includes addition, deletion and modification of LOCs. However, the computation of CHANGE in a evolution study usually excludes comments and blanks.

IPM - Instructions per Method

This metric is computed by the proportion between the total number of instructions (i.e., NBI) and the total number of methods (i.e., NOM) where NBI is a metric that counts the total number of byte-code instructions, ignoring comment lines and blank lines, and NOM is the number of methods metric defined before. The following formula is used to compute IPM metric:

$$IPM = NBI/NOM \tag{4.6}$$

4.3 Code Quality Popular Tools

Kotlin is a new programming language with respect to other languages like Java. Tools to measure the code quality metrics can be found easily for the popularity and fame of Java. On the other hand, code quality of Kotlin applications may be harder to find. However, the existence of tools for Object Oriented programming languages made it easier. In this section we discuss different popular tools used to measure the quality of both Kotlin and Java. Some special cases for popular tool that is used to measure the quality of Kotlin applications is also defined.

4.3.1 SonarSource

SonarQube

SonarQube empowers all developers to write cleaner and safer code. It is a tool that covers more than 25 different programming languages(Java, Kotlin,swift,js , etc.). SonarQube is an open source tool, produced by SonarSource so that using static analysis, a developer can measure Reliability, Security and Maintainability of all the languages used in the application. SonarQube provides hundreds of static code analysis rules with a consistent set of metrics.[27] Main features of SonarQube can be summarized by the following:

- **Detect Tricky Issues**: such as bugs, code smells, security venerability as for all execution paths.
- **Release Quality Code**: Quality Gates tell you at every analysis whether your code is ready to release.
- **Enhance Your Workflow**: Non-disruptive code quality checks overlay your workflow and intelligently promote clean builds.

Code Analyzers

SonarSouce provided a set of code analyzers for different programming languages, with these code analyzers, its possible to measure deep code metrics by finding the trickiest and nastiest bugs and quality issues using static code analysis techniques. SonarKotlin and SonarJava provide a static code analyzers that has a great coverage of well-established quality standards according to the official website of SonarSource.[28].

4.3.2 UNDERSTAND

UNDERSTAND is a closed source tool used to collect metrics about the code analysis, without a result the the code is 'good' or 'bad'. This will leave the choice

to developers to decide using the data report that helps the team to interpret what they think the best for the software. Using command line, graphical interface, or the API, the result metrics can be extracted. Different metric categories can be measured using UNDERSTAND code analysis tool:

- **Complexity Metrics** i.e. CC - McCabe's Cyclomatic Complexity and WMC - Weighted Method Count
- **Dimensional Metrics :** i.e. LOCs - Lines of Code and other metrics explained in the previous section.
- **Object Oriented Metrics :** i.e. NOCH - Number of Children , DIT - Depth of Inheritance Tree.

4.3.3 Codacy

Codacy is a tool that can be used to automate the code quality, it identifies issues through static code analysis. Codacy is a free tool for open source software, it notifies users on security issues, code coverage, code duplication and code complexity on every commit and pull requests. Codacy is a closed source tool that helps to speed up productivity, so that the focus during development time will be on programming by keeping technical dept under control. It is a flexible tool and can be integrated into workflow so it adapts to code review process[29].
As for our study interest was to find tools that supports both Kotlin and Java, so same standards are followed during code quality analysis. Codacy supports more than 30 programming languages including Java and Kotlin by providing the following features:

- **High-security standards :** Prevent critical issues from affecting applications by identifying vulnerabilities.
- **Code standardization :** Ensure that code quality is standardized by applying code patterns and getting notified on new issues.
- **Tailored to developer needs :** Save time by highlighting the broken standards and getting insights on how to solve them.
- **Integrated in your workflow :** Speed up the process by receiving notifications as pull request comments or on Slack.

4.3.4 Paprika

Paprika is a code analysis tool that detects different code smells in analysed Android applications. Paprika supports Object oriented and Android code smells for both

Kotlin and Java programming languages. Using the APK, SHA256 of the apk and without any need to use the source code of the application, Paprika is able to analyze mobile applications returning as output a csv file for each code smells. Paprika supports a list of 4 Object Oriented code smells and 13 Android code smells: [30]

- **Object-Oriented code smells:**
  1. Blob Class (BLOB)
  2. Swiss Army Knife (SAK)
  3. Long Method (LM)
  4. Complex Class (CC)
- **Android code smells:**
  1. Internal Getter/Setter (IGS)
  2. Member Ignoring Method (MIM)
  3. No Low Memory Resolver (NLMR)
  4. Leaking Inner Class (LIC)
  5. UI Overdraw (UIO)
  6. Invalidate Without Rect (IWR)
  7. Heavy AsyncTask (HAS)
  8. Heavy Service Start (HSS)
  9. Heavy Broadcast Receiver (HBR)
  10. Init OnDraw (IOD)
  11. Hashmap Usage (HMU)
  12. Unsupported Hardware Acceleration (UHA)
  13. Bitmap Format Usage (BFU)

However, analysis on Android applications must carefully keep a mix of smells related to complex/large code components (e.g., BLOB,SAK,LM and CC) as well as smells related Android application(e.g., NLMR,HAS,HSS,HBR,IOD and UOI). Kotlin and Java are both Object Oriented languages, we discuss some interesting object oriented code smells supported by paprika toolkit. In tables 4.1 and 4.2, we can see some object oriented and Android code smells respectively, with a short description for each one, the short description represents the reason of the occurrence of the code smell inside our application.

Code Smell	Description
BLOB Class(BLOB)	A Class with a large number of attributes and/or methods.
Swiss Army Knife(SAK)	An Interface with a large number of methods
Long Method(LM)	A Method which the number of instructions is higher to a given threshold.
Complex Class(CC)	A Class containing complex methods

Table 4.1: Object Oriented Code Smells

Code Smell	Description
No Low Memory Resolver (NLMR)	Activities missing onLowMemory() method.
Heavy AsyncTask (HAS)	Heavy operations are executed at the main thread in Async Task.
Heavy Service Start (HSS)	Heavy operations are executed at the main thread in Service.
Heavy Broadcast Receiver (HBR)	Presence of heavy or blocking operations in a broadcast receive.
Init OnDraw (IOD)	Allocations are made inside onDraw() routines.
UI Overdraw (UIO)	UI design consists of unneeded overlapping layers.

Table 4.2: Android Code Smells

4.4 Experiment Setup

In this section we show the setup of our experiment on a set of applications with more than two releases, to study the evolution of Android applications from Java to Kotlin. Using this experiment we will be able to answer the research question of what is the impact in terms of Quality of Android applications that made the transition from Java to Kotlin programming language.

4.4.1 Building Dataset of Android Application releases

The first challenge of this experiment was building the set of popular applications that had the transition from releases using only Java as production code to a release that is entirely using Kotlin.

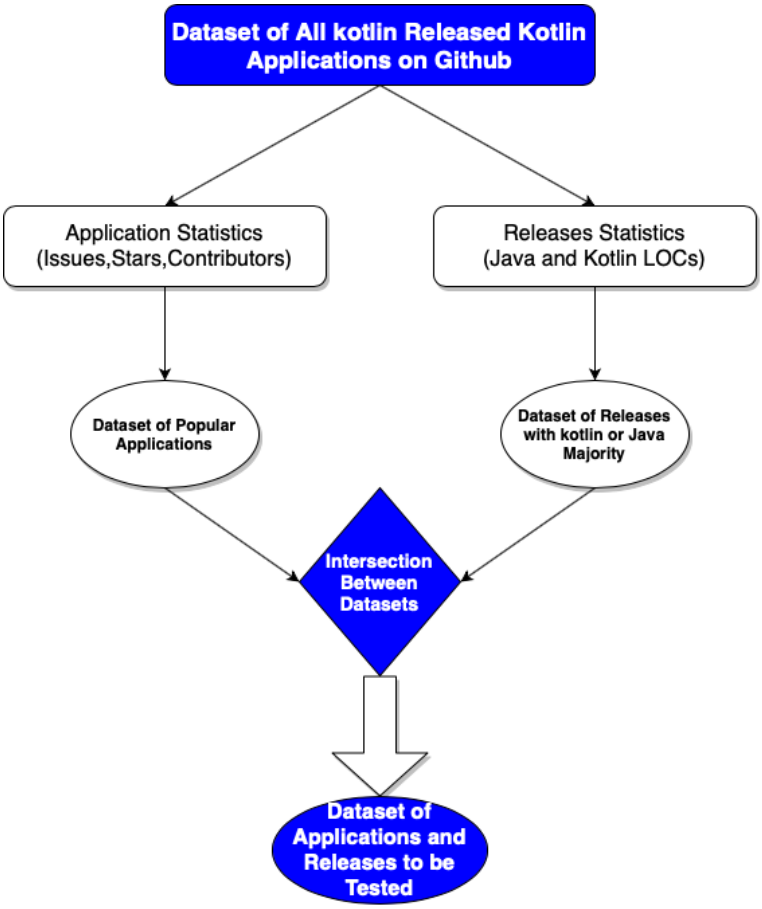

Figure 4.1: Applications selection for the study

Popular Applications on github

According to data mining in the second chapter, using the number of issues, stars and contributors of application as a measurement of popularity of an application so that we calculated a score using the equation bellow for all applications.

$$Popularity = (Issues + Contributors + Stars)/3 \tag{4.7}$$

In the equation above, we assumed that the three parameters have the same weight with respect to popularity and interactions with the repository on GitHub.

We can see in figure 4.2 a sample of the obtained dataset where we can notice that the number of stars was the key of high score applications:

	OwnerName	ProjectName	Issues	Contributors	Stars	Releases	score
1	shadowsocks	shadowsocks-android	19	39	28486	160	9514
2	square	leakcanary	33	76	24300	21	8136
3	afollestad	material-dialogs	37	87	17177	89	5767
4	Kotlin	anko	244	69	15510	26	5274
5	google	flexbox-layout	71	27	15130	24	5076
6	android	uamp	46	26	11421	3	3831
7	mikepenz	MaterialDrawer	5	94	10844	195	3647
8	android	sunflower	98	30	9819	7	3315
9	JakeWharton	RxBinding	43	39	9270	4	3117
10	AppIntro	AppIntro	36	75	8957	11	3022
11	JakeWharton	timber	48	40	7719	5	2602
12	inorichi	tachiyomi	423	70	6068	38	2187
13	android	topeka	8	18	5111	6	1712

(a) Fig1

	OwnerName	ProjectName	Issues	Contributors	Stars	Releases	score
14	agrosner	DBFlow	80	52	4734	32	1622
15	drakeet	MultiType	2	8	4542	35	1517
16	mikepenz	Android-Iconics	0	33	4509	96	1514
17	mozilla-mobile	fenix	1392	123	2951	56	1488
18	diogobernardino	williamchart	17	3	4285	3	1435
19	romannurik	muzei	31	24	3988	17	1347
20	Yalantis	Context-Menu.Android	7	8	3627	11	1214
21	RedApparat	Fotoapparat	66	25	3506	21	1199
22	airbnb	MvRx	31	1	3497	7	1176
23	kittinunf	fuel	44	67	3260	43	1123
24	vipulasri	Timeline-View	4	6	3073	9	1027
25	Triple-T	gradle-play-publisher	19	34	2945	38	999
26	mikepenz	FastAdapter	6	41	2923	131	990

(b) Fig2

Figure 4.2: Database table of popular applications

Applications with Transition from Java to Kotlin

The objective of this part of the study is to analyze the transition from Java to Kotlin applications in terms of code quality, so the study is done on different releases of applications.

Using CLOC tool, we counted the number of Kotlin and Java lines of code of all the set of releases of the popular applications mentioned before, so as an output of we obtained statistics about LOCs, then calculating the percentage of Kotlin and Java code for each release with respect to the production code helps to choose different releases of each application. One of the releases should be with the majority of Java as for the other selection should be with the majority

of Kotlin to study the code quality of applications during the two phases. As a point to start comparing releases, we supposed that tagged releases before October 2017 contains only Java code while new releases are mostly written in Kotlin programming language.

4.4.2 Analyzing Code Quality metrics

To perform our code quality analysis across Kotlin and Java application releases, the challenge of this section is to find a tool that is able to compute metrics of code quality for both languages(Java and Kotlin).

Starting from the set of releases of popular applications, to answer the research question of this chapter, we measured the quality metrics on every release in the new dataset that is composed of two releases of each application.

Selection of detecting tool

On a set of 15 applications we used an IntelliJ IDEA and IntelliJ Platform IDEs plugin called "MetricsReloaded" to calculate and measure code complexity metrics in Android applications It is applied on the source code of applications and it covers languages supported by the used IDE.

Our analysis was based on open source applications on GitHub, for this reason we selected Metrics Reloaded tool for static code analysis. However, the selected tool might take some time due to the manual work needed to analyze each application release. In table 4.3, we show an example of the output of McCabe's Cyclomatic Complexity metric on a sample application.

Method metrics | Class metrics | Package metrics | Module metrics | Project metrics

method	ev(G)	iv(G)	v(G)
null.onError(String)	1	1	1
null.onInit(int)	1	8	10
null.onItemClick(AdapterView<?>,View,int,long)	1	1	1
null.onPrepareActionMode(ActionMode,Menu)	1	2	2
null.onSensorChanged(SensorEvent)	1	1	2
null.onStart(String)	1	1	1
null.run()	1	1	1
Total	**2,594**	**3,966**	**4,330**
Average	1.15	1.76	1.92

Figure 4.3: CC metric applied on an application methods

Metrics Reloaded automates code metrics and offers more than 250 metrics that can be used by developers for static analysis. In section 4.2 we discussed popular code quality metrics that are used to maintain the source code during development.

For this reason, we measured the code quality of applications of the source code based on complexity metrics 4.2.1

Data Analysis

To answer the research question of this chapter we measured the code quality metrics on two different releases of the application. One release with the majority of Java and the other release with the majority of Kotlin. The comparison between both releases was necessary to measure the impact of the transition on the code quality of applications. We applied our analysis on 15 different applications chosen according to their popularity. We limited our study to 15 applications since, as it will be shown later, the analysis we carried out is not only computationally expensive, but also requires manual work on each application release separated.

Firstly, on project level, we measured the average McCabe's Cyclomatic Complexity(CC) metric. Secondly, on methods level we calculated the total number of statements in each method(STAT) and the ratio of lines of code for a method to the lines of code for it's containing class(RLOC). Finally, based on class level, we measured weighted Method Count(WMC).

To compare between the code quality of applications, we saved the results of the calculated metrics of each application as a ".csv" files, then using some queries we were able to measure the average with respect to all the dataset of applications, and to compare separately between applications releases before and after the transition to Kotlin.

Cyclomatic Complexity levels differs between Low Risk Program and Most complex and highly unstable method according to [**website:sourceforge**], Where they considered the value 10 as a threshold between acceptable code(low risk) and too complex code(higher risk). Other sources considered 15 as a threshold. In our study, Cyclomatic Complexity scores are categorized into the following levels:

- 1-10 = Low risk program
- 11-20 = Moderate risk
- 21-50 = High risk
- >50 = Most complex and highly unstable method

The Weighted Method Count metric is used as an indicator to how hard is it to maintain and develop a distinct class. The high value of WMC might be used as an indicator that the class is application specific and is responsible for more than one job. While lower WMC indicates a good abstraction. which is the main objective to handle complexity by hiding unnecessary details from the users. The threshold of the lower limit is 1, since each class must contain at least one method.

While no specific definition for the upper threshold, since it may differ between applications. We assumed that applications with higher WMC are more complex and harder to maintain as a reference.

Relative number lines of code (RLOC) measure the ratio of LOCs for a method over the LOCs of it's containing class. Higher value of RLOC indicates poor abstraction. It is expected that recent releases will have higher RLOC because of the new features added to applications.

Methods with large number of statements indicates that the code is harder to maintain. From developers point of view it is better to break down large methods into multiple small and focused methods.

4.5 Experiment Results

In this section we discus the observations as a result of static code analysis on different releases of applications and the impact of Kotlin on code quality of the applications.

To study the impact of Kotlin on the dataset of applications, first we started by comparing the collected metrics for each application release.

Analysis on Java Applications Releases

We started by collecting metrics on Android applications releases using Java as a main language of production code.

Our results shows that the average of the Cyclomatic Complexity in the set of applications included in our study is 2.36. While the upper limit of the average of CC in a single applications was 3.87. Using this results we concluded that based on the sample application, applications using Java where considered as low risk programs.

The maintainability of an application is proportional to the average Weighted Methods Count, so that an applications with high WMC is considered as hard to maintain. According to our results, the average summation of the complexity of all class methods implemented in our set of applications was 15.15. While the highest bound was 33.39 in "App 12". The number of statements in applications methods is another factor of maintainability. In our analysis we measured the average number of statements over all methods in each application. We found that over all the set of Java releases, the average of STAT metric is 5.34 and with an average below 10 in all applications included in our static code analysis.

The average percentage of RLOC over all the set of applications is 4.74%, which is considered an indicator of the level of abstraction. Low RLOC percentage represents good abstraction in our set of applications.

In table 4.3 we show the detailed results of each application included in our study based on the releases with the majority of Java over Kotlin code.

	Avg(CC)	WMC	RLOC	STAT
App 1	1.83	10.67	5.88	4.53
App 2	1.61	9.99	4.86	2.91
App 3	2.01	13.67	3.59	6.26
App 4	2.12	12.26	4.72	4.25
App 5	2.09	13.77	3.04	4.99
App 6	2.07	12.08	4.8	4.33
App 7	2.15	15.6	4.02	4.05
App 8	2.19	16.7	4.18	5.94
App 9	2.3	19.95	4.29	6.01
App 10	1.85	10.55	5.87	4.08
App 11	3.37	13.83	12.6	6.7
App 12	3.87	33.39	1.19	9.59
App 13	2.37	15.42	5.55	6.61
App 14	3.8	12.3	4.3	6.1
App 15	1.83	16.98	2.35	3.76
Avg	2.36	15.15	4.74	5.34

Table 4.3: Metrics on Applications releases with the majority of Java

Analysis on Kotlin Applications Releases

We applied the same code analysis on the source code of releases with the majority of Kotlin over Java. We found that the average of Cyclomatic Complexity over all applications is below 2.0, with 1.25 and 3.3 as lowest and highest limit of the average CC respectively, which in all cases indicates a low risk programs.

Our results shows that the average of WMC and STAT are 12.04 and 4.85 respectively. This factors will be used later to compare between maintainability of applications before and after using Kotlin. Finally, the result of RLOC metric is 6.41% , which represents a good abstraction.

In table 4.4, we can see the detailed results of each application included in our study on the releases using Kotlin as a majority with respect to the production code. We selected the last release expecting to find the best code quality due to the modifications after each release.

	Avg(CC)	WMC	RLOC(%)	STAT
App 1	1.8	8.39	7.83	3.59
App 2	1.66	10.07	4.8	2.98
App 3	2.09	20.14	2.83	7.15
App 4	1.28	4.8	2.53	2.56
App 5	1.25	6.77	7.74	3.01
App 6	1.73	8.71	12.54	4.23
App 7	1.35	7.44	9.63	3.91
App 8	2.3	20.4	2.7	6.8
App 9	2.45	22	2.83	7.09
App 10	2.1	10.55	2.93	3.1
App 11	3.17	10.46	5.1	6.82
App 12	1.38	9.25	10.44	2.85
App 13	1.75	23.69	1.7	3.37
App 14	3.3	12.3	3.1	4.6
App 15	2.02	8.06	19.55	5.73
Avg	1.97	12.04	6.41	4.58

Table 4.4: Metrics on Applications releases with the majority of Kotlin

Conclusion

To study the impact of Kotlin on Android applications, we compared the results of metrics calculated on applications releases before and after the deployment of Kotlin. Our findings showed that applications was characterized as low risk programs based on the results of the Cyclomatic Complexity. The average CC decreased by 16.52%, which is a good impact on the complexity of application after the transition to Kotlin.

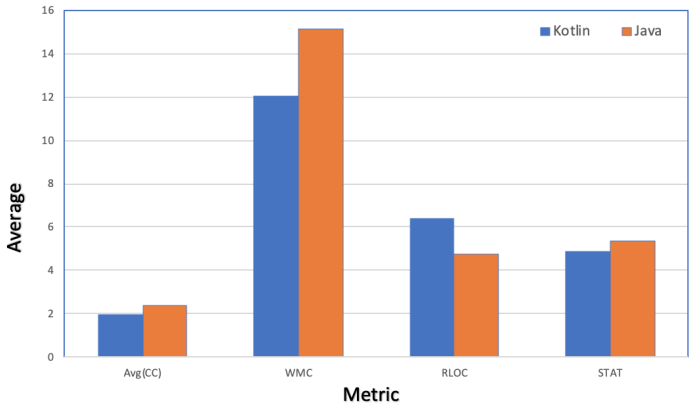

Figure 4.4: Java vs Kotlin metrics results

In terms of maintainability, the average WMC decline by 20.52% in application with the majority of Kotlin code. Same for the average of statements with respect to applications methods, that had decreased by 14.23% in the last release of applications. According to the results, Kotlin had a positive impact on the maintainability of applications, so that the decrease in the average of WMC and STAT indicates a less complex and easier to maintain applications. Knowing that the average of the factors used to indicate the maintainability of applications applied on Java majority releases showed acceptable results with a low average.

As expected, the only negative metric was RLOC, which had a slight increase in the percentage. This indicates that Kotlin majority releases included in our analysis had a poorer abstraction compared to Java majority. This can be justified by the addition of new features into the apps, which affects the relative number lines of code.

Chapter 5

Conclusion

In this section we discus the observations of our study. the study was based on inspecting open source Android application on GitHub, and then search the integration of Kotlin code in Android applications and the usage of Kotlin Development Practices in all the set of application. Finally, calculate the static analysis metrics on a sample set of applications to study the impact of Kotlin on the quality of applications converted the base code of releases from majority of Java to majority of Kotlin.

We found that most applications changed the majority of the base code into Kotlin, while a small ratio of applications are still using Java as a major language inside applications. From the evolution metrics, our results shows that the highest ratio of applications are still in the phase of transition since Java is still used in the production code of the applications. On the other hand, from the diffusion metrics, we found that the percentage of applications with majority of Kotlin lines of code over the production code is very high due to the filtering criteria of including only applications using Kotlin and measuring the metrics on the last release of applications. The results shows that applications developing teams are convinced by the benefit of transitioning to Kotlin, that was expected due to the features, safety and flexibility provided by Kotlin programming language.

We inspected applications source code searching for the usage of Kotlin development practices using regular expressions that represent each practice. Our results showed that practices like argument lists and safe casting are used rarely in applications with an average bellow 20%. While the average usage of unsafe casts with respect to applications exceeded the 70%. Comparing these results of the usage of Kotlin Dev. Practices and the transition of base code of applications, we concluded that applications made a huge transition but without the expected usage of the best development practices. The transition from Java to Kotlin without taking advantage of development practices, may be because of automatically converting the codebase. These tools reduce the cost of development while a human-based

tweaks are needed after the conversion. The lack in checks after conversion leads to a misuse of the Kotlin programming language.

Using "MetricsReloaded" plugin provided for IntelliJ IDEA and IntelliJ Platform IDEs, we were able to analyse code quality metrics on different releases of applications. Applications code quality is subjective, different tools might use different metrics to identify the quality of applications. In our analysis we measured metrics related to the complexity of applications. Our calculations showed that the transition to Kotlin improved the quality of applications in terms of complexity and maintainability.

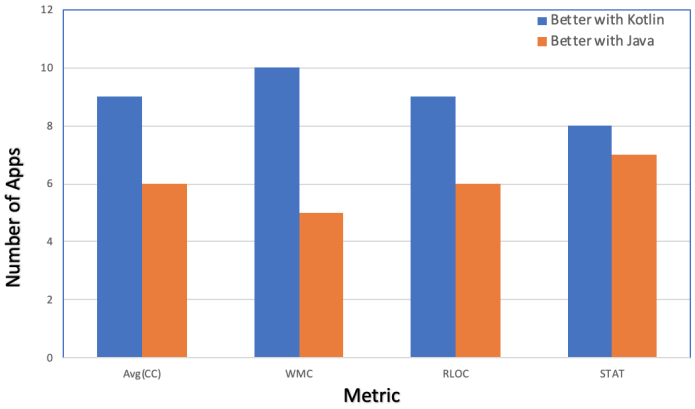

Figure 5.1: Java vs Kotlin: Number of improved applications

We found that 9/15 applications had a positive change in the average of Cyclomatic Complexity. 10 out of 15 applications showed an improvement in the WMC metric. and 8 of 15 shows lower number of statements per method. According to our results, more than half of the applications had an improvement in terms of maintainability of applications.

Another factor on code quality is the relative number lines of code which indicates the level of abstraction. applications in our set showed a higher result of RLOC after the transition to Kotlin. However, 9/15 applications improved the ratio of LOCs for a method over the LOCs of it's containing class.

Chapter 6

Threats To Validity

6.1 Construct Validity

Threats concern the relation between theory and observations. Our goal was to analyze the usage and impact of the transition to Kotlin on open source Android application on GitHub.

6.2 Internal Validity

Filtering and Cloning Android Mobile Applications

In section 2.2, we discussed the inclusion and exclusion criteria to filter Android applications from the full list of the repositories that featured Kotlin on GitHub. Due to the high number of applications, we were not able to check manually the repositories to exclude the tools and empty projects. We started by the procedure of Deleting all repositories that doesn't have a manifest file and a at least one call to the function SetContentView() and we supposed the rest of repositories are Android applications. this procedure may miss some exclusion of tools that includes any sample of "AndroidManifest.xml".

Applications Popularity

One of our objectives was to trace the popularity of applications. We used parameters like contributors, issued and stars to measure the popularity. Our data was collected using GitHub API's that doesn't provide the date of the assignment of the star on the repositories. We assumed that the assigned parameters are the indication of popularity and interactions with the repository on github.

Kotlin Dev. Practices

In our analysis, we calculated the average according to the occurrence and usage of development practices in applications. We measured the percentage of applications with at least one occurrence of every regular expression over the total number of applications. Some applications might have a high percentage of usage of practices. This approach may be affected by the high percentage of applications that are ignoring the Kotlin practices.

Code Quality metrics

The quality of application can be measured using different metrics in different domains. In our study, we focused on the code complexity, analyzing the source code of applications. Other metrics could result to a different definition of the quality of applications. Our study was done on a sample data set of applications. The results may differ in case we study a bigger set of applications.

6.3 External Validity

Validity of Inspection Tools

To compare the percentage of Kotlin and Java in applications and inspect the Kotlin practices, we used "CLOC" and "rg" tools. Our results are fully dependent on the efficiency of the used tools. Any bug in one of the tools could affect the analysis of diffusion and evolution metrics, as for the usage of Kotlin dev. practices in the applications.

Validity of MetricsReloaded Tool

MetricsReloaded is an automated code metrics for IntelliJ IDEA and IntelliJ Platform IDEs. It's computes metrics depending on the source code of applications. The results of our code analysis are dependent on the correctness of the tool.

Chapter 7

Future Work

In this section we show some work that can be done in the future. Due to the time limitation (fetching real data is usually time consuming), Some experiments have been left. Deeper analysis on different dataset of applications using different tools to have more details about applications state before and after the transition.

I would have liked to try some experiments about commits and watch APIs during development of the functions in Chapter 2, but for the lack of time(i.e. it might take days to finish fetching APIs due to GitHub rate limit and the big number of applications). In chapter 3 we focused on the main features. However, Investigating about other Kotlin practices may help to show better percentage about the usage of Kotlin new features. In chapter 4, Increasing the number of analyzed projects and including closed source applications may be interesting. Due to manual work needed to analyze each application release, and the time needed to finish the statistics we were not able to try this work. With respect to code quality smells inspected, there are various code smells that can be included to obtain more accurate results about the quality of applications. We chose on two releases to solve evolution metrics.

This thesis has been mainly focused on the popularity of Kotlin applications with respect to the percentage of Kotlin LOCs and development practices. Future research on closed source applications with matching applications with the popularity on Play Store might extend the explanations of the impact of the transition of Kotlin from users side.

Future research could examine different code analysis tools on applications APKs, to answer questions about code quality with respect to different code smells.

It is a question of future research to investigate about the cost of the transition from Java to Kotlin, and using the results of this study we can conclude the impact with respect to the cost of the transition. Base on that we can answer the question if changing the language of the codebase of android applications from Java to Kotlin worth the cost?