PhD Thesis

Université Polytechnique Hauts-de-France
and from INSA Hauts-de-France

Subject:

Computer Science

Presented and defended by **Bruno GÓIS MATEUS**

On **March 26, 2021**, Valenciennes

Doctoral School:

Sciences Pour l'Ingénieur (ED SPI 072)

Research team, Laboratory:

Département d'Informatique
Laboratory of Industrial and Human Automation control, Mechanical engineering and Computer Science (LAMIH UMR CNRS 8201)

Towards high-quality Android applications development with Kotlin

JURY

Committee President
- Káthia MARÇAL DE OLIVEIRA. Professor at Université Polytechnique Hauts-de-France.

Reviewers
- Guilherme HORTA TRAVASSOS. Professor at Federal University of Rio de Janeiro.
- Jacques KLEIN. Professor at University of Luxembourg.

Examiner
- Dalila TAMZALIT. Associate Professor at Université de Nantes.

Supervisor
- Christophe KOLSKI. Professor at Université Polytechnique Hauts-de-France.

Co-Supervisor
- Matias MARTINEZ. Associate Professor at Université Polytechnique Hauts-de-France.

Colophon

Doctoral dissertation entitled “Towards high-quality Android applications development with Kotlin”, written by Bruno GÓIS MATEUS, completed on May 4, 2021, typeset with the document preparation system LaTeX and the yathesis class dedicated to theses prepared in France.

Thèse de doctorat

Université Polytechnique Hauts-de-France
et l' INSA Hauts-de-France

Discipline :

Informatique

Présentée et soutenue par **Bruno GÓIS MATEUS**

Le **26 mars 2021**, à Valenciennes

École doctorale :

Sciences Pour l'Ingénieur (ED SPI 072)

Equipe de recherche, Laboratoire :

Département d'Informatique
Laboratory of Industrial and Human Automation control, Mechanical engineering and Computer Science (LAMIH UMR CNRS 8201)

Vers un développement d'applications Android de haute qualité avec Kotlin

JURY

Présidente du jury
- Káthia MARÇAL DE OLIVEIRA. Professeure à l'Université Polytechnique Hauts-de-France.

Rapporteurs
- Guilherme HORTA TRAVASSOS. Professeur au Federal University of Rio de Janeiro.
- Jacques KLEIN. Professeur à l'University of Luxembourg.

Examinatrice
- Dalila TAMZALIT. Maître de conférences HDR à l'Université de Nantes.

Directeur de thèse
- Christophe KOLSKI. Professeur à l'Université Polytechnique Hauts-de-France.

Co-encadrant
- Matias MARTINEZ. MCF à l'Université Polytechnique Hauts-de-France.

The Université Polytechnique Hauts-de-France and the INSA Hauts-de-France neither endorse nor censure authors' opinions expressed in the theses: these opinions must be considered to be those of their authors.

Keywords: Android development, Kotlin, adoption, evolution, migration, machine learning

Mots clés : développement Android, Kotlin, adoption, évolution, migration, apprentissage automatique

This thesis has been prepared at

Département d'Informatique
Laboratory of Industrial and Human Automation control, Mechanical engineering and Computer Science (LAMIH UMR CNRS 8201)
Université Polytechnique Hauts-de-France
Le Mont Houy
F-59313 Valenciennes Cedex 9

The roots of education are bitter, but the fruit is sweet.

Aristotle

Abstract

In recent years, with more than 3 million applications on its official store, Google's Android has dominated the market of mobile operating systems worldwide. Despite this success, Google has continued evolving its operating system and its toolkits to ease application development. In 2017 Google declared Kotlin as an official Android programming language. More recently, during the Google I/O 2019, Google announced that Android became 'Kotlin-first', which means that new API, libraries, and documentation will target Kotlin and eventually Java and Kotlin as preferred language to create new Android applications.

Kotlin is a programming language that runs on the Java Virtual Machine (JVM) and it is fully interoperable with Java because both languages are compiled to JVM bytecode. Due to this characteristic, Android developers do not need to migrate their existing applications to Kotlin to start using Kotlin in these applications. Moreover, Kotlin provides a different approach to write applications because it combines object-oriented and functional features. Therefore, we hypothesize that the adoption of Kotlin by developers may affect different aspects of Android applications' development. However, one year after this first announcement, there were no studies in the literature about Kotlin. In this thesis, we conducted a series of empirical studies to address these lacks and build a better understanding of creating high-quality Android applications using Kotlin.

First, we carried a study to measure the degree of adoption of Kotlin. Our results showed that 11% of the studied Android applications had adopted Kotlin. Then, we analyzed how the adoption of Kotlin impacted the quality of Android applications in terms of code smells. We found that the introduction of Kotlin in Android applications initially written in Java produces a rise in the quality scores from 50% to 80% according to the code smell considered. We analyzed the evolution of usage of features introduced by Kotlin, such as *Smart cast*, and how the amount of Kotlin code changes over applications' evolution. We found that the number of instances of features tends to grow throughout applications' evolution. Finally, we focused on the migration of Android applications from Java to Kotlin. We found that 25% of the open source applications that were initially written in Java have entirely migrated to Kotlin, and for 19%, the migration was done gradually, throughout several versions, thanks to the interoperability between Java and Kotlin. This migration activity is challenging because: a) each migrated piece of code must be exhaustively tested after the migration to ensure it preserves the expected behavior; b) a project can be large, composed of several candidate files to be migrated.

In this thesis, we present an approach to support migration, which suggests, given a version of an application written in Java and eventually, in Kotlin, the most *convenient* files to migrate. We evaluated our approach's feasibility by applying two different machine learning techniques: classification and learning-to-rank. Our results showed that both techniques modestly outperform random approaches. Nevertheless, our approach is the first that proposes the use of machine learning to recommend file-level migrations. Therefore, our results define a baseline for future work. Since the migration from Java to Kotlin may positively impact the application's maintenance and that migration is time-consuming and challenging, developers may use our approach to select the files to be migrated first. Finally, we discuss several research perspectives opened by our results that can improve the experience of creating high-quality Android applications using Kotlin.

Keywords: Android development, Kotlin, adoption, evolution, migration, machine learning

Résumé

Ces dernières années, avec plus de 3 millions d'applications sur sa boutique officielle, Android de Google a dominé le marché des systèmes d'exploitation mobiles dans le monde entier. Malgré ce succès, Google a continué à faire évoluer son système d'exploitation et ses kits d'outils pour faciliter le développement des applications. En 2017, Google a déclaré Kotlin en tant que langage de programmation Android officiel. Plus récemment, pendant le Google I/O 2019, Google a annoncé qu'Android devenait 'Kotlin-first', ce qui signifie que de nouvelles API, bibliothèques et documentations cibleront en priorité Kotlin, et éventuellement Java et Kotlin, comme langage préféré pour créer de nouvelles applications Android.

Kotlin est un langage de programmation qui s'exécute sur la machine virtuelle Java (JVM) et il est entièrement interopérable avec Java car les deux langages sont compilés en bytecode JVM. En raison de cette caractéristique, les développeurs Android n'ont pas besoin de migrer leurs applications existantes vers Kotlin pour commencer à utiliser Kotlin dans ces applications. De plus, Kotlin propose une approche différente pour écrire des applications car il combine des fonctionnalités orientées objet et fonctionnelles. Par conséquent, nous émettons l'hypothèse que l'adoption de Kotlin par les développeurs Android peut affecter différents aspects du développement des applications Android. Cependant, un an après cette première annonce, il n'y avait aucune étude dans la littérature sur Kotlin. Dans cette thèse, nous avons mené une série d'études empiriques pour combler ces lacunes et développer une meilleure compréhension de la création d'applications Android de haute qualité à l'aide de Kotlin.

Tout d'abord, nous avons réalisé une étude pour mesurer le degré d'adoption de Kotlin. Nos résultats ont montré que 11% des applications Android étudiées avaient adopté Kotlin. Ensuite, nous avons analysé l'impact de l'adoption de Kotlin sur la qualité des applications Android en termes de défauts de code. Nous avons constaté que l'introduction du code Kotlin dans les applications Android initialement écrites en Java produit une augmentation des scores de qualité de 50% à 80% selon les défauts de code considérés. Nous avons analysé l'évolution de l'utilisation des fonctionnalités introduites par Kotlin, telles que *Smart cast*, et comment la quantité de code Kotlin change au fil de l'évolution des applications. Nous avons constaté que le nombre d'instances de fonctionnalités a tendance à augmenter tout au long de l'évolution des applications. Enfin, nous nous sommes concentrés sur la migration des applications Android de Java vers Kotlin. Nous avons constaté que 25% des applications open source initialement écrites en Java ont complètement migré vers Kotlin, et pour 19%, la migration s'est faite progressivement, sur plusieurs versions, grâce à l'interopérabilité entre Java et Kotlin. Cette activité de migration est difficile car : a) chaque morceau de code migré doit être testé de manière exhaustive après la migration pour s'assurer qu'il préserve le comportement attendu ; b) un projet peut être énorme, composé de plusieurs dossiers candidats à migrer.

Dans cette thèse, nous présentons une approche de prise en charge de la migration, qui propose, étant donné une version d'une application écrite en Java et éventuellement, en Kotlin, les fichiers les plus pratiques à migrer. Nous avons évalué la faisabilité de notre approche en appliquant deux techniques d'apprentissage automatique différentes : la classification et l'apprentissage par rang. Nos résultats ont montré que les deux techniques surpassent légèrement les approches aléatoires. Toutefois, notre approche est la première à proposer l'utilisation du machine learning pour recommander des migrations au niveau des fichiers. Par conséquent, nos résultats définissent une base de référence pour les travaux futurs. Comme la migration de Java vers Kotlin peut avoir un impact positif sur la maintenance de l'application et que la migration est longue et difficile, les développeurs peuvent utiliser notre approche pour sélectionner les fichiers à migrer en premier. Enfin, nous discutons de plusieurs perspectives de recherche ouvertes par nos résultats qui peuvent améliorer l'expérience de création d'applications Android de haute qualité à l'aide de Kotlin.

Mots clés : développement Android, Kotlin, adoption, évolution, migration, apprentissage automatique

Acknowledgments

I would like to express my gratitude to all the people who have contributed to this thesis's realization. First, I would like to thank Sylvain Lecomte, for offering me the opportunity to carry out this doctoral work. An exceptional thanks to my co-supervisor, Matias Martinez, thank you for several hours of daily meetings, for many pieces of advice and discussions that helped me achieve this work. I also want to thank you for your moral support that helped me overcome the Ph.D. challenges. I truly appreciate all your efforts to make these three years an enjoyable experience for me. I have learned a lot from you. I also want to thank my advisor, Christophe Kolski, who was essential in the final straight. Thanks for being so kind and for accepting this challenge. Your experience gave me confidence. Besides my advisor and co-advisor, I would like to thank the members of my thesis committee: Guilherme Horta Travassos, Jacques Klein, Dalila Tamzalit and Káthia Marçal de Oliveira, for their time and feedback. I wish to show my gratitude to the current and former members of LAMIH team. Thanks for the amusing conversations at lunch, the generous pots, and for the journées du doctorant. I would like to thank the Région Hauts-de-France immensely for the financial support that made this research possible. Last but by no means least, thanks to my family and friends who have provided me moral and emotional support in my life.

Contents

Abstract xiii

Résumé xiv

Acknowledgments xv

Contents xvii

List of Tables xix

List of Figures xxi

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Problems . . . 2
  - 1.2.1 Problem 1 - The lack of knowledge about the adoption of Kotlin . . . 2
  - 1.2.2 Problem 2 - The impact in the quality of Android applications . . . 3
  - 1.2.3 Problem 3 - The evolution of Kotlin code . . . 3
  - 1.2.4 Problem 4 - Migrating applications from Java to Kotlin . . . 3
- 1.3 Thesis Contribution . . . 4
  - 1.3.1 Contribution 1 - The study about the adoption of Kotlin . . . 4
  - 1.3.2 Contribution 2 - Showing the impact of the adoption of Kotlin on the quality of Android applications . . . 4
  - 1.3.3 Contribution 3 - The evolution of Kotlin code in Android applications . . . 4
  - 1.3.4 Contribution 4 - An approach to assist the migration of Android applications 5
- 1.4 Outline . . . 5

2 State of the Art 7
- 2.1 Kotlin . . . 7
  - 2.1.1 History of Kotlin . . . 7
  - 2.1.2 Android development using Kotlin . . . 8
  - 2.1.3 Literature about Kotlin . . . 10
  - 2.1.4 Summary . . . 11
- 2.2 Quality of Android applications . . . 12
  - 2.2.1 Identification of code smells in mobile applications . . . 12
  - 2.2.2 Analysis over time of code smells . . . 14
  - 2.2.3 The impact of programming languages on the presence of code smells . . . 14
  - 2.2.4 Evolution patterns on Android Applications . . . 15

2.2.5 Summary . . . 16
2.3 Software maintenance and evolution . . . 16
2.3.1 Programming language evolution . . . 17
2.3.2 Programming language migration . . . 18
2.3.3 Summary . . . 20
2.4 Machine learning applied to software engineering . . . 20
2.4.1 Classification applied to software engineering . . . 20
2.4.2 Learning-to-rank applied to software engineering . . . 25
2.4.3 Summary . . . 27
2.5 Conclusion . . . 27

3 Measuring the adoption of Kotlin by Android developers 29
3.1 Study of the adoption of Kotlin . . . 30
3.1.1 Looking for Kotlin-based Android applications . . . 30
3.1.2 Analysis method . . . 32
3.1.3 Results . . . 35
3.2 Study of The proportion of Kotlin code in Android applications . . . 36
3.2.1 Applications analyzed in the study . . . 37
3.2.2 Analysis method . . . 37
3.2.3 Results . . . 38
3.3 Threats to Validity . . . 38
3.3.1 Internal . . . 39
3.3.2 External . . . 39
3.4 Summary . . . 39

4 Measuring and Comparing the quality of Android applications written in Kotlin 41
4.1 Study design . . . 42
4.1.1 Tool selection . . . 42
4.1.2 Code smell selection . . . 43
4.2 Comparing the quality of Kotlin-based and Java-based Android applications . 45
4.2.1 Applications analyzed in the study . . . 45
4.2.2 Analysis method . . . 45
4.2.3 Results . . . 46
4.3 Measuring the impact on the quality of introducing Kotlin . . . 50
4.3.1 Applications analyzed in the study . . . 50
4.3.2 Defining a quality model . . . 50
4.3.3 Training a quality model . . . 51
4.3.4 Analysis method . . . 51
4.3.5 Results . . . 53
4.4 Threats of validity . . . 56
4.4.1 Internal . . . 56
4.4.2 External . . . 56
4.5 Summary . . . 57

5 Analyzing the evolution of Kotlin code in Android applications 59
5.1 Analyzing the code evolution of Android applications . . . 60
5.1.1 Applications analyzed in the study . . . 60
5.1.2 Code evolution trends . . . 60
5.1.3 Analysis method . . . 62

5.1.4 Results . . . 63
5.1.5 Threats to Validity . . . 66
5.2 The usage and evolution of Kotlin features . . . 66
5.2.1 Study Design . . . 67
5.2.2 The adoption of Kotlin features . . . 70
5.2.3 The introduction of Kotlin features . . . 74
5.2.4 The usage evolution of Kotlin features . . . 75
5.2.5 Discussion . . . 81
5.2.6 Threats to Validity . . . 82
5.3 Summary . . . 83

6 Applying machine learning to assist the migration of Android applications 85
6.1 Study design . . . 88
6.1.1 Approach . . . 88
6.1.2 Projects analyzed in the study . . . 91
6.1.3 Feature extraction . . . 92
6.2 Research Question 9 . . . 95
6.2.1 Model training . . . 95
6.2.2 Evaluation . . . 96
6.2.3 Results . . . 97
6.3 Research Question 10 . . . 98
6.3.1 Model training . . . 99
6.3.2 Evaluation . . . 99
6.3.3 Results . . . 100
6.4 Discussion . . . 101
6.5 Threats to Validity . . . 102
6.5.1 Construct validity . . . 102
6.5.2 Internal validity . . . 102
6.5.3 External validity . . . 102
6.6 Summary . . . 103

7 Conclusion and perspectives 105
7.1 Summary of contributions . . . 105
7.2 Short-term perspectives . . . 106
7.2.1 Kotlin bad practices and code smells . . . 106
7.2.2 Power consumption on Kotlin-based Android applications . . . 106
7.2.3 Feature engineering for assisted migration . . . 106
7.3 Long-term perspectives . . . 107
7.3.1 Test generation for migration . . . 107
7.3.2 Automatic software refactoring for migration . . . 107
7.3.3 Cross-platform development using Kotlin . . . 107
7.4 Final words . . . 108

Bibliography 109

A Publications 131
Published: . . . 131
To submit: . . . 131

B Kotlin Features **133**
B.1 Type inference . . . 133
B.2 Lambda expressions and Anonymous functions . . . 135
B.3 Inline Function . . . 135
B.4 Null-safety . . . 135
B.5 When expression . . . 136
B.6 Default argument . . . 136
B.7 Named argument . . . 137
B.8 Smart cast . . . 137
B.9 Data classes . . . 139
B.10 Range expression . . . 139
B.11 Extension function . . . 139
B.12 String template . . . 139
B.13 Delegation . . . 140
B.14 Destructuring declaration . . . 140
B.15 Operator overloading . . . 140
B.16 Singleton . . . 142
B.17 Companion Object . . . 142
B.18 Infix function . . . 143
B.19 Tail-recursive function . . . 143
B.20 Sealed class . . . 143
B.21 Type alias . . . 144
B.22 Coroutine . . . 145
B.23 Contract . . . 145
B.24 Inline class . . . 145

List of Tables

2.1 Studies about Kotlin in the literature. 12

3.1 Steps executed to build a recent version of AndroidTimeMachine. 32
3.2 Classification of applications according to their programming language. 35

4.1 Paprika supported code-smells. The column 'Considered' shows the 10 code smells studied in our work (✓) and the 7 not studied (×). 43
4.2 Percentage of Android applications affected by code smell. An app a is affected by a code smell s if a has at least one instance of s. 47
4.3 Ratio comparison between Kotlin and Java. The column 'Cliff's δ' shows the difference between the smell median ratio of pure Kotlin and Java applications: negative values mean that a smell affects fewer entities in Kotlin than in Java. . 49
4.4 Changes on quality scores after introducing Kotlin. 53

5.1 Classification of Android applications according to the evolution trend of Kotlin and Java source code. 63
5.2 Kotlin features and their release version. 67
5.3 This table shows for each Kotlin feature X (rows) and for each evolution formula f (columns) the number of applications where the formula f better *describes* the evolution of the feature X. The last row, Total, shows how many features were better described by a trend (column) 80

6.1 List of collected features grouped by category. 94
6.2 Results of the data extraction. For 1% of GitHub projects the tool failed during the extraction. 94
6.3 The precision (Pr), recall (Re), and accuracy (Acc) of the different machine learning models and their training and test datasets. ↑ indicates that a metric has improved when compared with the empirical study using only $Android_{j2k}$ dataset. Analogously, ↓ indicates that a metric has decreased. 98
6.4 Mean Average Precision (MAP) at K of a random approach and LambdaMART. . 101

7.1 The short and long-term perspectives of our thesis. 108

B.1 Kotlin features and their release version. 134

List of Figures

2.1 The evolution of the number of questions about Kotlin on the StackOverflow over time. The bars in the graph are grouped by color according to the current Kotlin version in that period. 8
2.2 Timeline of studies published focused on Kotlin. 10

3.1 The steps needed to create the dataset target of this study. 33
3.2 Distribution of number of versions (apk) per application. 34
3.3 Our pipeline to classify Android applications (Section 3.1.2.). 34
3.4 Distributions between Kotlin and Java applications and versions. 36
3.5 To answer our RQ_2, we retrieved the corresponding code repository (e.g., a git repository) of each FAMAZOA's application. Then, for each repository, we executed CLOC to identify the proportion of Kotlin code. 37
3.6 Distribution of applications according the percentage of Kotlin code. 59.43% of Kotlin applications have more than 80% of source code written in Kotlin. 38

4.1 Steps followed to compare Java and pure Kotlin applications in terms of code smells. 45
4.2 Steps followed to measure the impact of introducing Kotlin on applications' quality. 52
4.3 Evolution of quality scores based on CC smell along the version history. 54
4.4 Three cases of classification of quality trend evolution *before* and *after* the introduction of Kotlin code. 55

5.1 Steps executed to perform the analysis of the code evolution trends of Android applications. 61
5.2 Evolution trends of Kotlin and Java code. Each plot presents the evolution of Kotlin and Java code along the history of one single application where the x-axis corresponds to the commits and the y-axis corresponds to the amount of code (i.e., lines). 64
5.3 Steps executed in out study about the use of Kotlin features. 68
5.4 Percentage of applications that use a feature. Each bar corresponds to a feature and contains on top the number of applications that use that feature. 71
5.5 Kotlin features normalized. 72
5.6 Distributions of the number of days between the first Kotlin commit and the commit that introduces the first instance of a feature. 75
5.7 Example of evolution trends. The x-axis shows the evolution of an application, i.e., commits, and the y-axis shows the number of occurrences of a feature. . . . 77

6.1 Evolution of the number of lines of Java and Kotlin along with the Duolingo application's migration process adapted from [45]. Axis Y shows the amount of code (LOC) written, and axis X the time from 2014. 86
6.2 Evolution of amount of Java and Kotlin code (LOC) from real open source application that was migrated completely in one single commit. 86
6.3 This figure shows how our approach applies classification and learning-to-raking to recommend file-level migrations. In this example, a given project has five files and according to the technique applied our approach suggests a set of files to be migrated. 89
6.4 In the development phase, we first collect several projects. Then, we filter projects that have migration. Later, we extract features from files and classify files between migrated or not migrated. Finally, using these information we train a model using projects with migration. 90
6.5 The creation of the training dataset for classification. Given P and its list of commits with migration, we checkout the source code for each commit. Then, we extract the features for all project files. Later, this information is formatted. In this format, each file becomes a row in the dataset, and each column contains information related to one feature. There is also one column that indicates if that file was migrated or not. 96
6.6 For each commit in the training dataset, our approach generates a ranking with all project files. This ranking is compared to an ideal ranking created from the corresponding commit, where the migrated files are in the top. In this example, we set $k = 2$, so only the first two positions of the rankings are considered in the evaluation. 100

Chapter 1

Introduction

1.1 Context

The development of applications for mobile devices started more than 10 years ago, but since the iPhone AppStore, a website that allows users to download applications, opened in July 2008, there has been exponential growth in mobile application development [274]. Over that time to the present day, mobile development platforms have evolved, and today one platform dominates this market, Google's Android, leading mobile operating system worldwide in July 2020 with a 74.6 percent share [252].

Since the first release of Google's mobile operating system, developers have been developing applications mostly using Java and, in some specific scenarios, using C++. Developers use the official IDE (Integrated Development Environment), named Android Studio, and a SDK (Software Development Kit) provided by Google to build Android applications. The SDK compiles the Java code into Dalvik bytecode, which is then packaged on an *apk*. Then, developers submit those apks to application stores, such as the official, named Google Play Store, which currently has more than 3 million applications in its official store [17]. Android users can install applications into their Android devices by downloading those apks directly from the applications stores.

Despite the Android operating system's success, different development approaches and frameworks have emerged to ease mobile applications' development during the last years [200, 180]. For instance, approaches such as PhoneGap/Cordova [16] from Adobe, Xamarin from Microsoft [192], React-Native from Facebook [81], and more recently Flutter from Google [100], aim at facilitating the development of multi-platform mobile applications by allowing developers to write applications using a non-native programming language and then to obtain a version of a native version for each platform (Android and iOS). Meanwhile, Google and Apple continue evolving their development toolkits to build native applications to avoid mobile developers migrating to such third-party development frameworks.

In an attempt to ease mobile programming practice, in June 2014, Apple released Swift, a modern, multi-paradigm language that combines imperative, object-oriented, and functional programming for developing iOS applications [231]. Similarly, in 2017 Google announced the Kotlin programming language as an officially supported language for Android development [53]. Kotlin is a pragmatic programming language that runs on the Java Virtual Machine (JVM) and it is fully interoperable with Java because both Java and Kotlin code is compiled to JVM bytecode. Consequently, it is possible to mix Kotlin and Java code in the same application, call Kotlin code

from Java code, and the opposite. Kotlin provides a different approach to write applications because it combines object-oriented and functional features, some of them not present in Java or not available for Android development [57].

The announcement made by Google promoting Kotlin as an official Android language kicked off a change in the development environment of the most popular operating system of mobile devices. More recently, during the Google I/O 2019, Google announced a big change. They declared that the Android became 'Kotlin-first', which means that new API, libraries, and documentation will target Kotlin and eventually Java [107], and also declared Kotlin as preferred language to create new Android applications [69]. Therefore, now developers have two options of official programming language for writing applications: Java and Kotlin.

Android applications that are high-rated on the Google Play Store are updated more frequently to exploit the latest features provided by the newest versions of the Android SDK [260]. As a consequence of the fact that the Android platform has become 'Kotlin-first', developers and companies would need to adopt Kotlin to keep their applications updated with the new platform's features.

Due to the interoperability between Kotlin and Java, to use Kotlin for creating Android applications, developers have roughly three possible approaches: 1) they can add new functionalities written in Kotlin and maintain the existing Java code, 2) they can replace portions of existing Java code with Kotlin code, or 3) they can migrate the Java code entirely to Kotlin. Therefore, Android developers do not need to migrate their applications completely as it would happen when migrating other legacy code e.g., from COBOL to Java, because it is possible to evolve these applications using both languages.

1.2 Problems

Previous works have shown that software written with multiple programming languages is more defect proneness [144]. Moreover, the simple fact of adopting another programming language may affect software quality [230]. Therefore, we hypothesize that the adoption of Kotlin by Android developers may affect different aspects of Android applications' development. However, in January of 2018, when we started this research, there were no studies about Kotlin in the literature.[1] Consequently, there was no knowledge about the impact of adopting Kotlin.

This thesis aims to address these lacks and build a better understanding of creating high-quality Android applications using Kotlin. We believe this understanding is helpful for the research community, developers, and practitioners working on mobile applications. It is worth noting that there was no study about Kotlin in the literature when we started this thesis.

To build this understanding, this thesis addresses different research problems. We detail in the following these problems and the motivation behind our focus on them.

1.2.1 Problem 1 - The lack of knowledge about the adoption of Kotlin

Kotlin became an official Android programming language in 2017, but until 2018 (this thesis started in February 2018), there was no study about Kotlin in the literature. Consequently, there was no scientific evidence that Kotlin has been used to build Android applications. Therefore, before trying to understand the possible consequences of adopting Kotlin by Android developers, it was necessary to verify whether Kotlin has been adopted to measure the degree of adoption and identify such applications to conduct further studies.

[1] The first study about Kotlin was published in September 2018 [84].

1.2.2 Problem 2 - The impact in the quality of Android applications

Since mobile applications' emergence as new mainstream software systems, researchers were interested in bad development practices in these frameworks, a.k.a. code smells [109]. Code smells are well-known in the object-oriented development community as poor or bad practices that negatively impact software maintainability [87]. Previous studies assessed the impact of code smells on application performance [116], power consumption [43, 197, 215] and quality in terms of code smells [115]. However, none of these studies have targeted Android Kotlin-based applications. Therefore, there is no knowledge about how the adoption of Kotlin impacts the applications' quality.

1.2.3 Problem 3 - The evolution of Kotlin code

As Kotlin is fully interoperable with Java, it implies that a developer can introduce Kotlin code into their Java-based applications without migrating that existing code. Thus, an application written initially in Java can evolve, for instance, by adding Kotlin gradually. Moreover, Kotlin combines object-oriented and functional features, which brings the development of Android applications programming language features not provided by Java such as *Coroutines* (a feature that provides a new way of writing asynchronous, non-blocking code). However, to the best of our knowledge, there is no study in the literature about the evolution of Kotlin code and the usage of Kotlin features by Android developers. Therefore, we do not know which Kotlin features developers are using or whether they are using features not available in Java. However, this lack of knowledge negatively affects four audiences: *i)* researchers are not aware of the research gaps (i.e., the actual unsolved problems faced by the developers) and thus miss opportunities to improve the current state of the art, *ii)* language and library designers do not know if the developers effectively use the programming constructs and APIs they provide or are rather misused or underused, *iii)* tool builders do not know how to tailor their tools, such as recommendation systems and code assistants, to the developers' actual needs and practices when using Kotlin, *iv)* developers are not aware of the good and bad practices related to the use of Kotlin features [186].

1.2.4 Problem 4 - Migrating applications from Java to Kotlin

Historically Android applications have been written in Java since 2008. With the announcements that stated Kotlin as an official Android programming language, Google changed the Android development environment. Moreover, Google decided that new Android features, APIs, and documentation will target Kotlin firstly. Therefore, developers and companies that want to keep updated with the Android platform's new features may need to work with Kotlin. Nevertheless, developers do not need to migrate all Java code, and consequently, their applications, because Kotlin is interoperable with Java. Developers may add Kotlin code as much as they desire or even migrate their applications gradually. A gradual migration means that a Java-based application is converted to Kotlin throughout its version. Thus, an application migrated gradually has at least three versions considering the presence of Java and Kotlin: 1) A version containing only Java code, 2) a version containing Java and Kotlin code, and 3) a version containing Kotlin code. The migration process is challenging because it is necessary to ensure that the application's behavior remains unchanged in addition to code translation. Therefore, it is a time-consuming task, even for a big company. [45]. However, to the best of our knowledge, the currently available tool, the auto-converter from Android Studio, only helps developers in code translation. It does not help developers decide which file to migrate first, which is especially important to gradually migrate an application.

1.3 Thesis Contribution

To address the aforementioned problems, we conduct in this thesis a set of studies that contribute to the research about the development of Android applications using Kotlin. We summarize these contributions in the following subsections.

1.3.1 Contribution 1 - The study about the adoption of Kotlin

The first contribution of this thesis is a study about the adoption of Kotlin by Android developers. In this study, we propose a methodology for detecting applications written in Kotlin. Using our methodology, we analyze the source code and the binary files of 2 167 Android applications and their 19 838 versions. This study shows that 11% of the studied Android applications have adopted Kotlin. Moreover, among applications that adopted Kotlin, it highlights that most applications have at least 80% of their codebase written in Kotlin.

1.3.2 Contribution 2 - Showing the impact of the adoption of Kotlin on the quality of Android applications

The second contribution of this thesis is an empirical study that shows the impact of the adoption of Kotlin on Android applications' quality in terms of the occurrence of code smells [99]. We use PAPRIKA [117], a static analysis tool capable of identifying code smells on the JVM byte-code, to perform this analysis. This study covers 4 Android-specific, 6 object-oriented code smells and 2 167 Android applications. Using PAPRIKA, we analyze the proportion of applications affected by code smells, and we perform a more in-depth analysis comparing the proportion of applications' entities affected by code smells. The results showed that Java applications have more entities affected by all object-oriented code smells (4) and by one Android code smell. Moreover, it shows that the introduction of Kotlin code in Android applications initially written in Java produces a rise in the quality scores from 50% to 80% of the Android applications according to the code smell considered. This fact suggests that the migration of applications to Kotlin may reduce the occurrences of code smells and improve its quality.

1.3.3 Contribution 3 - The evolution of Kotlin code in Android applications

The third contribution of this thesis comprises two empirical studies focused on the evolution of Kotlin code into Android applications. First, we investigate how the amount of Kotlin code changes over the evolution of Android applications. In particular, in this study, we investigate whether developers use Kotlin to create new applications from scratch or to add new functionalities into existing Java-based applications, or even if they are migrating Java code to Kotlin. Then, in the second study, we focus on the 26 Kotlin features introduced, such as *Smart cast*.[2] In particular, we explore four aspects of features usage: *i)* **which** features are adopted, *ii)* **what** is the degree of adoption, *iii)* **when** are these features added into Android applications for the first time, and *iv)* **how** the usage of features evolves along with applications' evolution. The first study results show that once Kotlin is introduced into an application, it tends to become the repository's dominant language. Moreover, it also shows that 25% of the studied Android applications that introduced Kotlin have entirely migrated. Therefore, these findings suggest that the amount of Kotlin code in Android applications tends to increase in the next years, likewise the number of migrated applications. The second study results show

[2]Smart cast is a Kotlin feature that may avoid the use of explicit cast operators. More details are available in Appendix B

that Android developers are using all the studied Kotlin features that are not available in Java, including the experimental ones. Furthermore, it shows that the number of instances of features tends to grow throughout applications' evolution. These results confirm that the community of Android developers has been using Kotlin consistently.

1.3.4 Contribution 4 - An approach to assist the migration of Android applications

The last contribution of this thesis is an approach to assist Android developers in migrating applications to Kotlin. This approach applies machine learning techniques to suggest file-level migrations. File-level migration is important since converting classes in a different order may give different results [1]. Moreover, deciding which file migrate first arbitrarily may cause some re-work [204]. To evaluate our approach, we built a large-scale corpus of open source projects that migrated Java files to Kotlin and applied two different machine learning techniques: classification and learning-to-rank. Our results showed that both techniques outperforms a random approach. The best classification model, Random Forest, showed a 16% improvement in accuracy when compared with a random approach. On the other hand, our learning-to-rank model outperforms a random approach with at least 50%, but this result is still limited compared to a hypothetical perfect model. To the best of our knowledge, our study is the first one the investigates the performance of classification and learning-to-rank models on file-level migrations. Therefore, these results establish the initial baseline on the suggestion of file migration

1.4 Outline

The remainder of this dissertation is composed of 5 chapters as follows:

Chapter 2 - State of the art: This chapter provides an overview of previous work that focuses on Kotlin, the quality of Android applications, software maintenance, and evolution, focusing on programming languages and machine learning techniques applied to software engineering.

Chapter 3 - The adoption of Kotlin by Android developers: This chapter presents a study on the adoption of Kotlin by Android developers. This chapter is a revised version of the following paper:

- Bruno Góis Mateus and Matias Martinez. "An empirical study on quality of Android applications written in Kotlin language". In: *Empirical Software Engineering* 24 (6) (2019), pp. 3356–3393. DOI: 10.1007/s10664-019-09727-4

Chapter 4 - Kotlin and the quality of Android applications: This chapter describes an empirical study that investigates the impact of the adoption of Kotlin in Android applications' quality in terms of code smells. This chapter is a revised version of the following paper:

- Bruno Góis Mateus and Matias Martinez. "An empirical study on quality of Android applications written in Kotlin language". In: *Empirical Software Engineering* 24 (6) (2019), pp. 3356–3393. DOI: 10.1007/s10664-019-09727-4

Chapter 5 - The evolution of Kotlin code: This chapter shows a study about the evolution of Kotlin code in Android applications and the usage of Kotlin features by Android developers. This chapter is a revised version of the following paper:

- Bruno Góis Mateus and Matias Martinez. "An empirical study on quality of Android applications written in Kotlin language". In: *Empirical Software Engineering* 24 (6) (2019), pp. 3356–3393. DOI: 10.1007/s10664-019-09727-4
- Bruno Gois Mateus and Matias Martinez. "On the Adoption, Usage and Evolution of Kotlin Features in Android Development". In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM '20. Bari, Italy: Association for Computing Machinery, 2020. ISBN: 9781450375801. DOI: 10.1145/3382494.3410676

Chapter 6 - Using machine learning to assist migration of Android applications: This chapter presents a study investigating the feasibility of using a machine learning model to assist developers who want to migrate their applications to Kotlin.

Chapter 7 - Conclusion: This chapter presents the conclusion of this thesis and the short and long-term perspectives for our research.

We list our publications in Appendix A.

Chapter 2

State of the Art

In this chapter, we review the works that are closely related to our research topic. As Kotlin is the center of this work, Section 2.1 presents Kotlin and the literature about Kotlin and its use to develop mobile applications. Section 2.2 presents studies about the quality of mobile applications. Section 2.3.1 reports studies about the evolution of the usage of programming languages. Finally, Section 2.4 reviews studies that apply machine learning classification and learn-to-Ranking to solve software engineering problems.

2.1 Kotlin

In this section, we first present the history of Kotlin. Then, we highlight how Android developers can use Kotlin to create applications. Finally, we describe the literature about Kotlin.

2.1.1 History of Kotlin

Kotlin is a statically typed programming language that combines object-oriented and functional features, some of them not available in Java, for instance, *Smart cast* (See Appendix B.8). Designed to be an industrial-strength object-oriented language and a 'better language' than Java, but still, be fully interoperable with Java code [134].

Kotlin's history started in July 2011, when JetBrains, a Czech software development company, unveiled Project Kotlin, a new language for the Java Virtual Machine (JVM), which had been under development for a year [149]. Less than one year later, in February 2012, JetBrains open sourced the project under the Apache 2 license [275]. However, only in February 2016, its first version, Kotlin 1.0, was officially released [31]. In 2017, Kotlin reached an important milestone, with two versions released that year, Kotlin 1.1 in February and Kotlin 1.2 in November. Kotlin was announced as the third programming language fully supported for Android, in addition to Java and C++, during the Google I/O conference in May [53]. This announcement brought the possibility for developers to use Kotlin instead of Java to write Android applications and, consequently, diffuse the use of Kotlin among Android developers. In October 2018, Kotlin 1.3, well-marked by Coroutines' addition for asynchronous programming, was released almost one year later. In the same year, Kotlin reached a remarkable level of popularity. It was the fastest-growing language on GitHub with 2.6 times more developers than in 2017 [97]. Later, in 2019, another announcement made by Google had a notorious impact on the environment of Android

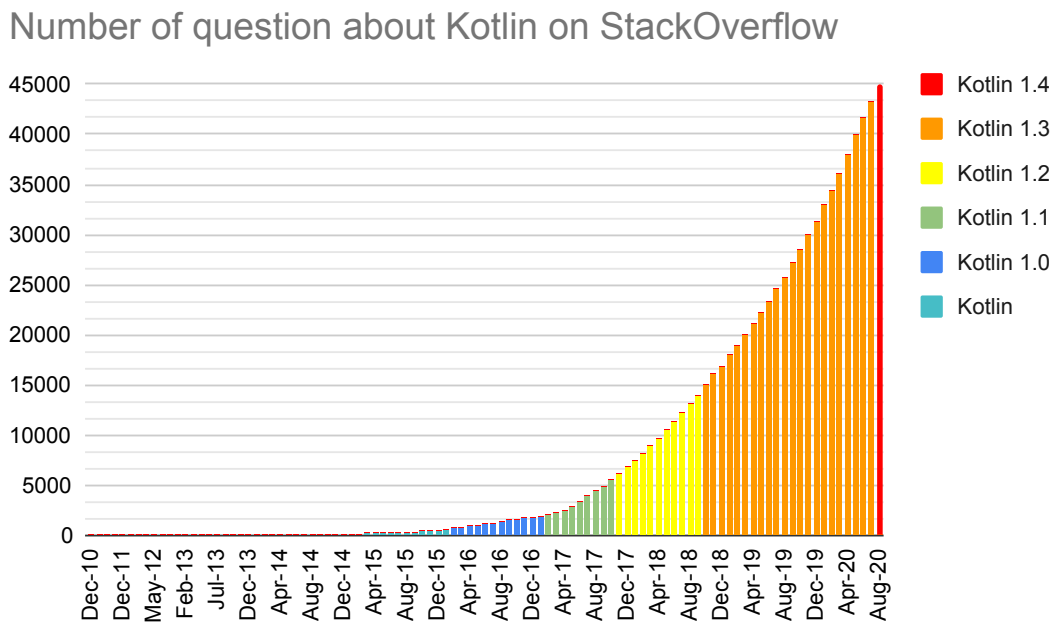


Figure 2.1: The evolution of the number of questions about Kotlin on the StackOverflow over time. The bars in the graph are grouped by color according to the current Kotlin version in that period.

development and the history of Kotlin. Google announced that Android became 'Kotlin-first', which means that new APIs and features will be offered first in Kotlin [107]. Moreover, Google started to advise developers to use Kotlin to create new applications and adopted Coroutines as the main solution for asynchronous programming on Android applications [72, 98].

All these facts helped Kotlin to become the fourth most loved programming language, according to the 2020 StackOverflow Developer Survey [210]. Moreover, as Figure 2.1 illustrates, Kotlin is attracting the interest of developers increasingly. The total number of questions about Kotlin on StackOverflow has been increasing continuously since Kotlin's first release.

Several aspects may have contributed to Kotlin's success, like the robust software company that develops it, Google's announcements, and its programming language features. Since some of these features are only found in Kotlin, we described them in Appendix B.

2.1.2 Android development using Kotlin

For decades, Java has been the foundation of innovative products for Android users for coders and programmers. However, Google, the Android owner, chose Kotlin as the alternative for Java for creating Android applications recently. Kotlin is both created and supported by JetBrains, IntelliJ's makers, the backbone of Android Studio, the official Android Integrated Development Environment (IDE) [64]. Therefore, to develop a mobile application using Kotlin, developers can use the same tools that Google provides for developing Android applications using Java: the Software Development Kit (SDK) and Android Studio.

Android Studio 3.0+ fully supports Android applications' development using Kotlin and pro-

```kotlin
import java.util.Calendar
fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) {
        calendar.firstDayOfWeek = Calendar.MONDAY
    }
}
```

Listing 1: Example of Kotlin code calling a Java class from the standard library. In this snippet inside the function calendarDemo, a Calendar (Java class) object is instantiated and later some of its properties are accessed.

```kotlin
//Kotlin code
@file:JvmName("DemoUtils")
package org.example
class Util
fun getTime() { /*...*/ }
```

```java
// Java code
package demonstration;

new org.example.Util();
org.example.DemoUtils.getTime();
```

Listing 2: In snippet on the left a Kotlin file defines a class named Util and a function named getTime. The snippet on the right shows how that Kotlin code is called using Java.

vides features such as auto-complete, debugging and refactoring. Kotlin is 100% interoperable with the Java programming language, so it is possible to call Java-based code from Kotlin, or call Kotlin from Java-based code [69], see Listing 1 and 2. Moreover, thanks to the interoperability between Kotlin and Java, using Android Studio, a mobile developer can: *1)* start a new Android project for developing an app using Kotlin from scratch, *2)* add new Kotlin files to an existing project already written in Java, or *3)* convert existing Java code to Kotlin. Because Kotlin compiles down to JVM bytecode and is fully interoperable with Java, developers or companies that want to migrate their applications from Java to Kotlin may do it gradually. Therefore, the development team can prioritize parts of the application to be migrated based on different criteria. Once the migration is completed, developers may exhaustively test these parts to verify whether it preserves the expected behavior.

Kotlin's concise syntax and ability to easily extend existing APIs lends itself to both new and existing Android APIs [64]. Core Android APIs and Jetpack components have been extended through the Android KTX extensions. KTX extensions provide concise, idiomatic Kotlin to Jetpack, Android platform, and other APIs [68].

All these factors motivate companies to adopt Kotlin for developing their applications. For instance, at Google, over 60 of Google's applications are built using Kotlin [69]. At the same time that Kotlin became more used among companies, it attracted researchers' interest, resulting in several studies about this promising programming language.

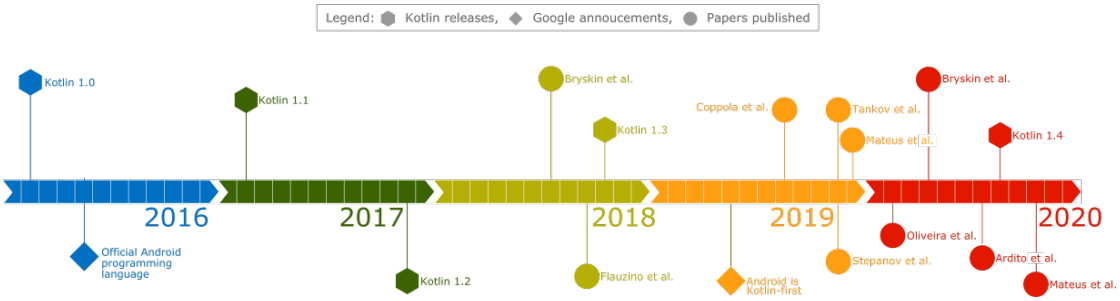


Figure 2.2: Timeline of studies published focused on Kotlin.

2.1.3 Literature about Kotlin

The interest of practitioners in Kotlin has been growing since 2015, as Figure 2.1 shows. This interest led to the production of many contents in different formats about this programming language. In 2017, the first books about Kotlin were published [125, 239, 198, 47, 267]. These books focus on different aspects of software development using Kotlin. On the academic side, the first studies were published in 2018, as Figure 2.2 illustrates. Furthermore, Figure 2.2 also shows that several studies were published after that. To present the current state of the art about Kotlin, in this Section, we cover all studies found in the literature focused on Kotlin.

The first work about Kotlin was published in September 2018 [84]. We remark our research started in January 2018 and we submitted our first study in July 2018, which was published in 2019 [99]. In that study, Flauzino et al. [84] compared Java and Kotlin applications in terms of the presence of code smells. We detail this study in Section 2.2. In the same year, Bryksin et al. [34], supported by researchers from JetBrains, presented a preliminary study about the detection of code anomalies on Kotlin code, which was further extended in 2020 [33]. They presented a method capable of detecting two types of code anomalies: *syntax tree anomaly* and *compiler-induced anomaly*. A *syntax tree anomaly* is a code fragment written in some way that is not typical for the programming language community. A *compiler-induced anomaly* is a code fragment that is not an anomaly in the syntax tree form but is an anomaly in the bytecode or vice versa. The authors applied their method on 47 751 repositories collected from GitHub and found 91 syntax tree anomalies and 54 compiler-induced anomalies. Then, they presented these anomalies to the Kotlin compiler developers to assess their relevance. 38 syntax tree anomalies and 31 compiler-induced anomalies were considered useful. Moreover, some of these anomalies were added into the compiler testing infrastructure as performance and correctness tests. Therefore, the authors concluded that the detected anomalies are useful and valuable for language development.

In 2019, Coppola, Ardito, and Torchiano [59] evaluated the transition of Android applications to Kotlin to understand whether the adoption of Kotlin impacts the success of an application. The authors mined all the projects from the F-Droid [103], a repository of Android open source applications. After finding the corresponding projects on the official Google Play Store and the GitHub platform, they statistically analyzed 1232 applications. They found that 19% of projects featured Kotlin code and that among those projects, the transition from Java to Kotlin was mostly fast and unidirectional. The authors also concluded that projects with Kotlin exhibited higher values for the rating and the number of downloads on Google Play Store, and the number of stars on GitHub.

The Kotlin compiler was also the target of researchers. Stepanov, Akhin, and Belyaev [253] focused on creating an automatic input reduction tool for the Kotlin compiler to simplify the

debugging process. The approach is based on a combination of Kotlin-specific transformations, program slicing, and Hierarchical Delta Debugging (HDD) [194]. Based on that approach, the authors have implemented a prototype tool called ReduKtor. After evaluating their tool, they concluded that to achieve high reduction quality, one must still employ language-specific transformations and general approaches, such as HDD.

Tankov, Golubev, and Bryksin [258] presented a framework for the development of web services named Kotless, Kotlin Serverless Framework. Kotless is a cloud-agnostic toolkit that interweaves the target application into the cloud infrastructure and automatically generates the necessary deployment code. Based on the internal use at JetBrains, the authors concluded that Kotless is a reliable, scalable, and inexpensive solution for a wide variety of web applications in a production environment.

Oliveira, Teixeira, and Ebert [205] performed a triangulation study to understand how developers are dealing with the adoption of Kotlin on Android development, their perception of the advantages and disadvantages related to its usage. They analyzed 9,405 questions related to Kotlin's development for the Android platform on StackOverflow. Moreover, they conducted qualitative research interviewing seven Android developers that use Kotlin to confirm and cross-validate their results. The authors conclude that developers seem to find Kotlin easy to understand and to adopt. Moreover, developers believe that Kotlin can improve code quality, readability, and productivity.

Ardito et al. [19] conducted a study to assess the assumed advantages of Kotlin concerning Java in the context of Android development and maintenance. To provide practical evidence that could help transition from Java to Kotlin, they conducted a controlled study with undergraduate students. They found that the usage of Kotlin apparently does not affect the maintainability with respect to Java when working on two small applications. Also, they found evidence that the adoption of Kotlin led to more compact code when developers were asked to develop new features for an ongoing software project. Finally, the authors concluded that most of the development promises using Kotlin are reflected by the code produced and the developers' perception.

2.1.4 Summary

We summarize in Table 2.1, the main research works that studied Kotlin. We observe that these studies have focused on different aspects of the usage of Kotlin. Coppola, Ardito, and Torchiano [59] and Oliveira, Teixeira, and Ebert [205] investigated the adoption of Kotlin by Android developers. Coppola, Ardito, and Torchiano [59] studied the impact of this adoption on the popularity of Android applications. On the other hand, Oliveira, Teixeira, and Ebert [205] mined questions and answers from StackOverflow to access the advantages and disadvantages of adopting Kotlin. Two studies focused on the maintenance of Android applications. In the first one, Flauzino et al. [84] compared Android applications written in Java and Kotlin in terms of code smells. In the second one, Ardito et al. [19] compared maintenance and development tasks performed on Android applications using Java and Kotlin.

Three studies focused on improving the Kotlin compiler [34, 33, 253]. Bryksin et al. [34, 33] investigated code anomalies in Kotlin and whether these anomalies could improve the Kotlin compiler. Stepanov, Akhin, and Belyaev [253] proposed a tool to perform input reduction, which simplifies the bug localization process. Finally, Tankov, Golubev, and Bryksin [258] proposed a framework for the development of web services. This framework is written in Kotlin, cloud-agnostic, and has been internally used by JetBrains.

Table 2.1: Studies about Kotlin in the literature.

Study	Purpose	Focus
Flauzino et al. [84]	Compare the occurrence of code smells between Java and Kotlin apps	Object-oriented code smells
Bryksin et al. [34] and Bryksin et al. [33]	Create an automated toolset capable of extract abnormal code fragments from a large Kotlin codebase,	Code anomalies
Stepanov, Akhin, and Belyaev [253]	Create a tool for the Kotlin compiler in order to simplify the debugging process	Bug localization and input reduction
Coppola, Ardito, and Torchiano [59]	Investigate Kotlin presence in open source projects	Diffusion, Evolution, Popularity Metrics
Tankov, Golubev, and Bryksin [258]	A framework for development of web services	Serverless applications
Oliveira, Teixeira, and Ebert [205]	Understand how developers are dealing with the adoption of Kotlin on Android development	Questions and answers
Ardito et al. [19]	To investigate advantages of Kotlin with respect to Java in the context of Android development and maintenance	maintenance and development tasks performed on Android applications

2.2 Quality of Android applications

Kotlin became an Android official programming language and, consequently, part of the ecosystem of mobile applications. Mobile applications are part of our daily lives, and they have attracted the attention of researchers. A variety of aspects of mobile applications, like the occurrence of code smells, have been investigated. Kent Beck coined the term *code smell* in the context of identifying quality issues in code that can be refactored to improve the maintainability of software [87]. Traditional code smells capture very general principles of good design. Consequently, specific code smells are needed to capture "bad practices" on software systems based on a specific platform, architecture, or technology [14]. In this context, studies about code smell of a specific domain have been conducted: object-relational mapping frameworks [50], mobile applications [269, 232, 117, 110], Cascading Style Sheets (CSS) [187] and Model View Controller (MVC) Architecture [14]. One of the goals of this work is to understand the impact of adopting Kotlin on mobile applications' quality. For that reason, in this section, we discuss the relevant literature that focuses on code smells to investigate mobile applications' quality.

2.2.1 Identification of code smells in mobile applications

The work of Reimann, Brylski, and Aßmann [232] was the pioneer regarding code smells specific to mobile applications, proposing a catalog of 30 quality smells dedicated to Android. To create this catalog, the authors mined different information sources: the official Android documentation, Google I/O talks, blogs from developers at Google or other companies, websites of discussion, like the StackOverflow, and discussions on the Android bug tracker.[1] To identify code smells from these resources, Reimann, Brylski, and Aßmann [232] used a three-step approach. In the gathering phase, official interfaces for querying the respective provider were used to download the available data into a local database. Then, in the phase of filtering, a preliminary selection of potentially relevant information was realized based on keywords as: "energy efficiency", "memory", and "performance", paired with issue keywords like "slow", "bad", "leak", and "overhead". Finally, the authors read the collected resources and created their catalog of code smells dedicated to Android. The authors also created a tool, named Refactory, to detect and

[1] https://source.android.com/setup/contribute/report-bugs

fix the occurrences of code smells from their catalog. Implemented as a plugin that extends the Eclipse platform, the tool can detect only nine quality smells from the catalog.

Hecht et al. [117] proposed a tooled approach, named Paprika, to identify object-oriented and Android-specific code smells from binaries (apk) of Android applications. Paprika is an open source tool that works at the level of bytecode. It is composed of three steps. In the first step, Paprika parses the apk to collect meta-data and code-metrics using the Soot framework [266] and its Dexpler module [21]. The information extracted in the first step is converted to a graph model. In the second step, this model is persisted using the graph database Neo4j.[2] Then, in the last step, using the Cypher query language[3], queries are used to detect code smells. When Paprika was first released and evaluated, it supported eight code smells, including Android-specific [117]. Currently, it supports 13 Android code smells.

Palomba et al. [214] proposed a detection tool, called AnDrOid Code smell detecTOR (ADOCTOR), to detect 15 Android code smells from the catalog of Reimann, Brylski, and Aßmann [232]. ADOCTOR relies on the analysis of the abstract syntax tree of Java code to identify code smells. The authors conducted an empirical study involving 18 Android applications to validate their proposed tool, and the results showed an average precision and recall of 98%.

Kessentini and Ouni [140] used a Multi-Objective Genetic Programming (MOGP) algorithm [12] to generate rules automatically for detecting code smells in Android applications. The proposed approach takes as input a set of Android-specific code smell examples from multiple applications. It uses a MOGP algorithm to find the best set of rules covering most of the expected Android code smells. A rule is a combination of quality metrics with threshold values. The authors reported that the generated detection rules identified 8 Android-code smell types with average correctness higher than 82% and an average relevance of 77%, according to the feedback of 27 Android developers.

The previous studies have focused on the detection of Android-code smells. However, other studies focused on the occurrence of well-known object-oriented code smells, for example Long Method and Large Class [86]. Linares-Vásquez et al. [163] presented a large-scale study over 1 343 Java Mobile Edition [206] applications to understand the relationships among smells, application domains, and quality. They used Defect dEtection for CORrection (DECOR) [195], an approach that allows the specification and automatic detection of code and design smells, to detect 18 object-oriented code smells using bytecode analysis of applications from 13 different domains. Their results showed that some code smells are more common in specific categories of Java mobile applications.

Mannan et al. [174] compared the presence of object-oriented code smells in 500 Android applications and 750 desktop applications in Java. They concluded that there is no major difference between these two types of applications concerning the density of code smells. However, they observed that the distribution of code smells on Android applications is more diversified than for desktop applications.

Khalid, Nagappan, and Hassan [141] analyzed the presence of object-oriented code smells to investigate whether they impact Android applications' rating. The authors examined the relationship between the application ratings from end-users with code smells detected using FindBugs [121],a tool that applies static analysis to find bugs in Java code, from 10 000 Android applications. They concluded that three categories of FindBugs warnings appear significantly more in low-rated apps.

While the previous studies have focused on identifying code smells using structural information extracted from source code, Palomba et al. [212] presented Historical Information for Smell deTection (HIST), an approach to detect smells based on change history information mined from

[2] http://neo4j.com/

[3] http://neo4j.com/ developer/cypher-query-language

versioning systems. HIST aims to detect five smells. The authors report that HIST has precision between 72% and 86% and recall between 58% and 100%. Comparing HIST with techniques and tools that do not consider historical information, the authors found that HIST outperformed them in terms of recall, precision, and F-measure. Furthermore, HIST was able to identify code smells that the other approaches could not. Finally, they concluded that there is a potential to combine historical and structural information for better smell detection.

2.2.2 Analysis over time of code smells

Historical information of systems was not only used to detect code smells. Unlike Palomba et al. [212], Tufano et al. [263] used historical information about changes in open source projects to investigated when code smells are introduced and the circumstances and reasons behind that introduction. Their empirical study over the commit history of 200 projects found that code artifacts are often affected by code smells since their creation. Moreover, they observed that developers tend to introduce smells mostly when implementing new features and enhancing existing ones.

Habchi, Rouvoy, and Moha [110] presented a large-scale empirical study investigating the evolution of mobile-specific (Android) code smells in the change history. Using their tool, named Sniffer, which relies on Paprika to detect code smells, they analyzed 180k instances of 8 types of Android code smells in 324 Android apps, covering 255k commits. The authors reported that, while Android code smells can remain in the codebase for years before being removed, it only takes 34 effective commits to remove 75% of them. Moreover, they found that Android code smells disappear faster in bigger projects with more commits, developers, classes, and releases.

Palomba et al. [213] presented a large-scale empirical study on the diffuseness of code smells and their impact on code change- and fault-proneness. They analyzed a total of 395 releases of 30 open source projects, considering 13 different code smells. They found that smells characterized by long and/or complex code (e.g., Complex Class smell) are highly diffused, and smelly classes have a greater change- and fault-proneness than smell-free classes.

Habchi, Moha, and Rouvoy [109] used historical information to conduct a large-scale empirical study about developers' role in the rise of mobile-specific code smells. After mining the change history of Android of 324 applications to track developers' contributions using their tool Sniffer, they concluded that the rise of Android code smells is not the responsibility of an isolated group of developers. Furthermore, most regular developers participated in the introduction of code smells. They also found that the ownership of code smells spread across developers regardless of their seniority.

2.2.3 The impact of programming languages on the presence of code smells

Habchi et al. [108] studied code smells in the iOS ecosystem, considering Swift and Objective-C languages, both used to write mobile applications for iOS, and how it is compared with Android smells. They proposed a catalog of 6 iOS-specific code smells based on information extracted from developers' feedback and the official iOS platform. To identify those code smells, they extended Paprika [117]. Analyzing 103 Objective-C applications and 176 Swift applications, they discovered that code smells tend to appear with the same proportion or only a slight difference in Objective-C and Swift. Furthermore, they analyzed 1 551 Android open source applications from F-Droid. They found that Android applications tend to contain more code smells than iOS applications in both languages (Objective-C and Swift), except for the Swiss Army Knife (SAK) smell, which appears in the same proportion for all languages.

Flauzino et al. [84] compared Java and Kotlin applications in terms of the presence of code

smells. They investigate the total of 100 projects, involving 50 Java and 50 Kotlin, and using five code smells: Data Class, Large Class, Long Method, Long Parameter List, and Too Many Methods, to verify whether these code smells are more common in Java applications. Applying an empirical study that involved more than 6 million lines of code, they found that on average Kotlin programs have fewer code smells than Java programs.

2.2.4 Evolution patterns on Android Applications

Several aspects of the evolution of Android applications were investigated in the literature.

Hecht et al. [115] used code smells to propose a fully automated approach that monitors mobile applications' evolution and assess their quality. Using Paprika [117], they mined 106 Android applications and 3 568 versions. To compute the software quality score, they created an estimation model using linear regression. Considering seven different code smells, they tracked the quality score of several versions of each application, and as a result, the authors identified five different quality evolution trends.

To better understand the evolution of mobile applications and find patterns in their evolution process, Zhang, Sagar, and Shihab [289] examined the applicability of three of Lehman's [152, 153] laws on mobile apps: continuing change, increasing complexity and declining quality. The law of continuing change says that systems must be continually adapted else they become progressively less satisfactory. The law of increasing complexity says that as systems evolve, their complexity increases unless work is done to maintain or reduce it. The law of declining quality says that the quality of systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes. To conduct this study, they extracted six evolution metrics from two Android applications and two desktop applications. Their findings showed that the law of continuing change and declining quality seems to apply for mobile applications. Moreover, they found that the desktop and mobile versions have different trends for the law of increasing complexity and the law of declining quality. Like Zhang, Sagar, and Shihab [289], Li et al. [158] conducted an empirical study focused on Lehman's laws. The authors examined long spans in the lifetime of 8 typical open-source mobile applications, covering 348 official releases, and several metrics were extracted to capture the characteristics of mobile applications. By observing the evolution trend of these metrics, they examined Lehman's laws to verify whether they still apply to mobile applications or not. Their results indicated that only a subset of Lehman's laws is still applicable to mobile applications. Moreover, the authors found that mobile applications' growth is non-smooth, and software instability increases with the addition of third-party libraries.

Beyond code smells, authors have identified evolution patterns of different issues related to Android applications' quality. Calciati and Gorla [39] investigated the evolution of permission requests across different releases of Android applications. The authors analyzed 14 000 releases of 224 Android applications to understand how requests change and how they are used. They identified a common trend where applications require more permissions over time.

Malavolta et al. [169] investigated the evolution of 6 maintainability issues, for example, Module Coupling and Duplication [270], along with Android applications' evolution. They statically analyzed 434 GitHub repositories of applications published on Google Play. Inspecting the density of maintainability issues, they identified 12 different evolution trends. They concluded that independently of the type of development activity and notwithstanding the issue type, maintainability issue density grows until it stabilizes. Moreover, the authors found that maintainability hotspots are independent of the type of development activity.

Calciati et al. [40] proposed a framework named Cartographer to analyze the evolution of Android applications. Cartographer extracts and shows various information such as how an

application uses sensitive data, which third-party libraries relies on, which URLs it connects, and more. Using this framework, the authors analyzed 235 applications with at least 50 releases. They concluded that Android applications tend to have more leaks of sensitive data over time.

Gao et al. [94] conducted an extensive empirical study focused on the maintainability and code complexity of mobile applications to understand how mobile applications' quality evolves. Relying on six metrics proposed by Chidamber and Kemerer [51], the authors investigated evolution trends of complexity applications from the AndroZoo [6, 161], a dataset of Android applications. Their results showed that Android applications usually become bigger during their evolution, and updates tend to add new classes. However, they conclude as well that complexity evolution is more like to increase or decrease wavily.

2.2.5 Summary

We reviewed in this section research work related to the topic of quality of mobile applications. This review showed many studies that focused on code smells (object-oriented and mobile-specific) to access mobile applications' quality. We also discussed studies that used historical information to: (i) identify code smells, (ii) investigate the survival of code smells, (iii) understand the diffuseness of code smells, and (iv) understand developers' role in the rise of code smells.

In this thesis, we are interested in understanding the impact of the adoption of Kotlin on the quality of mobile applications. For that reason, among the studies that focused on code smells, the ones more relevant to this thesis are: (i) Hecht et al. [117] because their proposed tool Paprika is capable of identifying code smells on Java and Kotlin applications since it works at the bytecode level, (ii) Hecht et al. [115] because they presented an approach to track applications quality based on the occurrence of code smells and (iii) Habchi, Rouvoy, and Moha [110] because they compared the impact of programming language on the presence of code smells. It is important to note that the study of Flauzino et al. [84], which is closely related to this thesis, is not listed among the most relevant study because it was not published when we started working on this topic.

Although several studies focused on the quality of mobile applications, there was no research about the adoption of Kotlin and its impact on mobile applications' quality. After analyzing the studies about mobile applications' quality, we review another facet of our topic in the upcoming section, which is the adoption and evolution of programming language features.

2.3 Software maintenance and evolution

Maintenance and evolution of application software consume a range high as 75-80 percent of a system's total life cycle cost [162]. This change process continues until it is judged more cost-effective to replace the system with a recreated version [153].

Along with software evolution, changes in software and its business, operational, and development environment will impact the software and conversely. For instance, whenever a new version of the programming, it is quite possible that programs that worked perfectly in a previous version of the language fail to function in the new version [190].

With the announcements that stated Kotlin an official Android programming language and that Android platform became 'Kotlin-first', Google changed the environment of Android development. After that, developers gained a choice between two programming languages, Kotlin and Java, for writing Android applications. At the same time, Google decided that new Android features, APIs and documentation will target Kotlin firstly. Therefore, developers and companies that want to keep updated with new features of Android platform need to work with Kotlin.

Kotlin has some unique features, like Coroutines, the recommended solution for asynchronous programming on Android [72]. Thus, developers who decide to use Kotlin need to learn a new programming language and new programming language features. Moreover, instead of adding some Kotlin code to applications, developers may choose to migrate their applications to Kotlin.

These possible scenarios were target of research that we present in the next sections. Section 2.3.1 presents studies about the evolution of programming languages. Section 2.3.1 highlights studies concerning the migration of software.

2.3.1 Programming language evolution

Popular programming languages evolve over time and one driver of evolution is a desire to simplify using languages in real-life projects. Therefore, evolving languages often address frequently encountered problems by adding language and library support to subsequent releases. Consequently, by using new features, programmers can express their intent more directly [226]. Simultaneously, programming language tends to become more complex because, over time, features are added but rarely removed for fear of breaking backward compatibility [209]. Thus, evolving programming languages can be problematic; evaluating the impact of changes is difficult, and it is often unclear how to effectively co-evolve software written in the language [265]. Moreover, when a new programming language feature is released, there is no guarantee that developers will use it. For that reason, in this section, we present studies that investigate the adoption of programming languages.

Several empirical studies about features of different programming languages have been conducted. Sutton, Holeman, and Maletic [254] developed a tool to identify the use of generic libraries in C++ projects to support a programmer's comprehension. Parnin, Bird, and Murphy-Hill [217, 216] conducted an empirical study to understand how Java generics[4] have been integrated into open source software by mining 40 popular Java programs' repositories. They found that one or two contributors often adopt generics and that generics reduce typecasts in projects.

Uesbeck et al. [264] investigated the impact of lambda expressions on the development, debugging, and testing effort in C++. In their experiment, two groups of developers, one using lambda expressions and one using iterators, had to solve four programming tasks. Analyzing the logs generated during the experiment, the authors conclude that participants spent more time with compiler errors and had more errors when using lambdas as compared to iterators, suggesting difficulty with the syntax chosen for C++.

Pinto et al. [224] studied the energy efficiency of Java's Thread-Safe Collections. They empirically investigated 16 collection implementations (13 thread-safe, 3 non-thread-safe) grouped under 3 commonly used forms of collections (lists, sets, and mappings). The authors concluded that simple design decisions could impact energy consumption and that different implementations of the same thread-safe collection can have widely different energy consumption behaviors.

Chapman and Stolee [48] studied the use of regular expressions in Python by conducting an empirical study of open source projects from GitHub. To understand when and how programmers use regular expressions, they surveyed 18 developers and statically analyzed 3 898 projects. They found six common behaviors that describe how regular expressions are often used in practice.

JavaScript's features were also the target of studies. Silva et al. [246] conducted a large-scale study to understand how class emulation is employed in JavaScript applications. The authors

[4]It is the enhancement to the Java type system allows a type or method to operate on objects of various types while providing compile-time type safety.

created a tool, JsClassfinder, capable of identifying class emulation by applying static analysis. They mined 50 popular applications from GitHub and found that 26% of projects do not use classes. Gallaba et al. [92] focused on the usage of *asynchronous callbacks* in JavaScript. The authors presented a set of program analysis techniques to detect instances of asynchronous callbacks and refactor such callbacks into promises (an alternative to asynchronous callbacks). These techniques were implemented in their tool named PromisesLand. They evaluated PromisesLand on 21 large JavaScript applications and found that it substitutes callbacks to promises correctly, with precision and recall of 100% and 83%, respectively.

Other studies investigated how the adoption of features evolves overtime. Pinto et al. [225] and Wu et al. [278] performed large empirical studies to investigate the usage of concurrency constructs in open source applications in Java and C++, respectively. Pinto et al. [225] have analyzed 2227 Java projects comprising more than 600 million lines of code. They found more than 75% of the latest versions of the projects either explicitly create threads or employ another alternative for concurrency control. Furthermore, they observed that efficient and thread-safe data structures, such as *ConcurrentHashMap*, are not yet widely used. Wu et al. [278] analyzed 492 concurrent open source applications, comprising 131 million lines of C ++ code. They found that small-size applications introduce concurrency constructs more intensively and quicker than medium-size applications and large-size applications. Moreover, the authors concluded that most projects do not move from third-party concurrency constructs to standard concurrency constructs.

Osman et al. [208][207] conducted an empirical study on long-lived Java systems to understand how exception usage changes as these systems evolve. The authors observed that the amount of error-handling code, the number of custom exceptions, and their usage in catch handlers and throw statements increase as projects evolve [208]. Moreover, they found that the domain, the type, and the development phase of a project affect the exception handling patterns [207].

Yu et al. [286] conducted an empirical study to evaluate the usage, evolution, and impact of Java Annotations. The authors analyzed 1 094 projects hosted on GitHub using Spoon, a tool for source code analysis and transformation for Java code [218]. They observed a strong relationship between using annotations and developer ownership, where developers with high ownership of a file are more likely to use annotations. To identify the uses overtime of Java's new features, Dyer et al. [77] analyzed 23 000 open source Java projects. They observed that all features are used even before their release date and that there were still millions of opportunities for use. Mazinanian et al. [186] focused on the evolution trend of one Java feature, Lambdas. They conducted an empirical study by statically analyzing the source code of 241 open source projects and interviewing 97 developers who introduced lambdas in their projects. As a result, they revealed an increasing trend in the adoption of lambdas.

2.3.2 Programming language migration

Over time, users of programming language might be impacted by a new release of existing programming language and its new features or a new programming language release. They must decide between using their preferred programming language and adopting their new features or adopting a new programming language. In Android development, developers faced a similar situation when Kotlin was announced as an Android platform's official programming language. While new Android developers may decide to adopt Kotlin, Android developers that are used to code in Java may decide between performing or not a language migration. For that reason, this section focus on programming language migration.

Martin and Müller [179] presented a structured approach for migrating C source code to

Java, minimizing manual intervention by software engineers. The authors reported that their 3 case studies were successful, but there was a need to conduct more case studies with larger programs to further validate their approach. Mossienko [199] presented an automated approach for source-to-source translation of Cobol applications into Java. Focusing on ensuring that the generated code will be maintainable, their approach does not guarantee complete functional equivalence of the generated code with the initial program. Therefore, some manual changes might be necessary, as the authors reported in the presented case of study.

El-Ramly, Eltayeb, and Alla [229] presented an experimental language transformer, J2C#, to automatically convert Java to C# using tree rewriting via functional rule-based programming. Marchetto and Ricca [175] defined a stepwise approach to help developers migrating a Java application into an equivalent service-oriented system. To evaluate their approach, they applied it to four simple Java applications. They concluded that the approach was successfully applied, but it needs to be applied to bigger and more complex applications for evaluating its scalability.

Colosimo et al. [56] presented Migration Environment for Legacy Information Systems (MELIS), an Eclipse plugin to migrate legacy COBOL programs to the web. To evaluate MELIS's effectiveness, they compared master students and professional developers' performance on completing migration tasks. The experiment revealed that software engineers' experience does not significantly affect the effort to migrate. Moreover, they reported that when MELIS is used, productivity improves seven or eight times compared to traditional development environments.

Zhong et al. [291] proposed an approach to assist code migration, called Mining API Mapping (MAM), that automatically mines how APIs of one language are mapped to APIs of another language. They compared the API mapping relations mined by their approach with manually written API mapping relations of Java2CSharp, a migration tool. They reported that compared with the mapping files of Java2CSharp, their mined mapping files show reasonably high precision and recall and include new mapping relations not covered by Java2CSharp. Moreover, they concluded that Java2CSharp could be improved by adopting API mapping relations mined using MAM.

Trudel et al. [262] presented C2Eif, a supporting tool (compiler), for source-to-source translation of C code into Eiffel. In their experiments, C2Eif translated completely automatically over 900,000 lines of 11 C code from real-world applications, libraries, and test suites, producing functionally equivalent Eiffel code. The authors concluded that C2Eif correctly translates complete applications and libraries of significant size and takes advantage of Eiffel's advanced features to produce safer code.

Nguyen et al. [201][202] introduced a data-driven approach that statistically learns the mappings between APIs from the source code of the same project written in C# and Java named Starminer. The authors empirically evaluated their approach and showed that it could mine a large number of API mappings with 87.1% accuracy. Compared against MAM, [291] it achieved higher precision (17.1%) and recall (28.6%). Moreover, they observed that when Java2CSharp uses API mappings mined using Starminer, it produces code with 4.2% fewer compilation errors and 6.0% fewer defects than with the mappings from MAM. Gu et al. [102] proposed a deep learning-based system for API migration named Deep API Migration (DEEPAM). DEEPAM is a learning architecture to learn joint semantic representations of bilingual API sequences from big source code data. Their experiment focused on 1-to-1 API mappings and reported that DEEPAM performs better than Starminer [201]. They concluded that deep learning in API migration is one step towards automatic code migration.

Malloy and Power [171] [172] investigated the degree to which Python developers are migrating from Python 2 to 3 by measuring the adoption of Python 3 features. Using their tool for syntax and feature recognition, the authors analyzed 51 Python applications hosted on GitHub. They concluded that developers were more often choosing to maintain backward compatibility

with Python 2 instead of exploiting the new features and advantages of Python 3.

Verhaeghe et al. [268] proposed an approach to help developers migrate the Graphical User Interface (GUI) of web-based software systems. To validate their approach, they developed a tool that migrates Java GWT applications to Angular. They evaluated their approach to an industrial application. The authors reported, they were able to model all the web pages of the application and that 26 out of 39 pages (66%) were migrated successfully.

Robillard and Kutschera [234] presented a case study of the migration of an interactive diagramming tool written in Java from the Swing Graphical User Interface framework to the more recent JavaFX framework. The authors reported five lessons about the discrepancies between expectations and reality in information discovery when migrating software between major frameworks: i) adapting to detail, ii) false friends, iii) feature gap, iv) feature blindness and v) hidden information. Moreover, for each lesson, they provided insights about how to better structure the information search in the context of migration frameworks.

2.3.3 Summary

In this section, we reviewed studies focused on the adoption of programming features and code migration. This review showed that several programming language features of different programming languages had been studied. However, Kotlin was not the target of these studies. Regarding studies about code migration, different approaches were proposed to help developers migrate from one programming language to another in different contexts, such as legacy systems. However, there is no consolidated solution to that problem. Moreover, we could not find any research about the migration of mobile applications. Consequently, we conclude that there is no study to support the migration of Android applications from Java to Kotlin.

2.4 Machine learning applied to software engineering

Machine learning techniques have been applied in different domains, including software engineering. In this Section 2.4.1, we present several studies that applied machine learning classification models to predict software aspects like vulnerabilities. Section 2.4.2 presents studies that applied learning-to-rank, another machine learning technique, to solve software engineering problems.

2.4.1 Classification applied to software engineering

Prediction is a core part of estimation, which is a crucial aspect of project planning [133]. Moreover, prediction depends mainly on historical internal and external quality attributes from completed projects [7]. Since machine learning techniques offer algorithms that can automatically enhance their performance through experience [35], several studies applied them to predict different software aspects. In this section, we present studies that applied machine learning classification to solve software engineering problems.

Koten and Gray [145] evaluated and compared the Naïve Bayes classifier [131] with regression-based models. Their results suggest that the Naïve Bayes model can predict maintainability more accurately than the regression-based models for one system, and almost as accurately as the best regression-based model for the other system. Moreover, they concluded that Naïve Bayes is indeed a useful modelling technique for software maintainability prediction, although further studies are required to realize the full potential as well as the limitation.

Elish and Elish [78] have empirically evaluated the capability of TreeNet [90, 91], a multiple additive regression trees algorithm, in predicting object-oriented software maintainability. The

authors compared TreeNet prediction performance against recently published object-oriented software maintainability prediction models: multivariate adaptive regression splines, multivariate linear regression, support vector regression, artificial neural network, and regression tree. The results indicated that when TreeNet is applied, prediction and accuracy improve or achieve competitive results compared to other models.

Romano and Pinzger [235] investigated various source code metrics' predictive power to classify Java interfaces into change-prone and not change-prone. More specifically, they compared the predictive power of the set of metrics defined by Chidamber and Kemerer [51], complexity metrics and two metrics to measure the external cohesion of Java interfaces. The authors performed a series of classification experiments with different machine learning algorithms, namely: Support Vector Machine, Naive Bayes Network and Neural Networks. Their results showed that most of the Chidamber and Kemerer metrics perform well for predicting change-prone concrete and abstract classes but are limited in predicting change-prone Java interfaces. Therefore, they concluded that interfaces need to be treated separately.

Al Dallal [3] empirically studied the impact of size, cohesion, and coupling on class maintainability. Using logistic regression [120], the authors explored the abilities of 19 measures, considered both individually and in combination, to predict class maintainability. They found that when the measures are combined, their models' abilities to predict class maintainability improve. Moreover, they concluded that prediction models based on the measures could help software engineers locate classes with low maintainability.

Kaur, Kaur, and Pathak [137] investigated whether a set of 23 metrics proposed by Meyer [191] and Spinellis and Jureczo [251] could be used for maintainability prediction. For that, they mined four different open source software repositories and calculated the correlation between these metrics and the number of changes made to the source code. Then, they built prediction models using classifiers: Naïve Bayes [131], bayes network, logistic regression [120], multilayer perceptron and random forests [29]. They concluded that accurate models could be constructed for software maintainability prediction using the set of metrics studied.

Elish, Aljamaan, and Ahmad [79] conducted three empirical studies on predicting software maintainability using ensemble methods. In each study, they developed different ensemble methods and compared their prediction performance in predicting software maintenance effort (study 1) and change proneness (study 2 and 3). They concluded that some ensemble methods provide more accurate or at least competitive prediction accuracy compared to individual models.

Kaur, Kaur, and Jain [136] investigated the relationship between object-oriented static software metrics, code smells, and change-prone classes. The authors compared machine learning techniques to predict software change-proneness using code smells and object-oriented metrics as predictor variables. They concluded that code smells are better predictors of change-proneness as compared to object-oriented software metrics.

Basgalupp, Barros, and Ruiz [23] applied an evolutionary algorithm called LExicographical Genetic Algorithm for Learning decision Trees (LEGAL-Tree) [22] for evolving decision trees tailored to predict software effort using a worldwide IT Company effort data set. They showed that evolutionary-based decision trees could outperform established approaches for decision-tree induction and traditional logistic regression.

Aljamaan, Elish, and Ahmad [4] proposed and empirically evaluated an ensemble of computational intelligence models for predicting software maintenance effort. They considered as ensemble constituent models, four popular prediction models (Multilayer Perceptron, Radial Basis Function Network, Support Vector Machines, and M5 Model Tree). The results confirm that the proposed ensemble technique provides more accurate prediction compared to individual models.

Khoshgoftaar et al. [142] investigated the usefulness of Classification And Regression Trees (CART) [30], a classification tree algorithm to predict fault-proneness system modules over several system releases. The authors considered forty-two software product, process, and execution metrics as candidate features predictors. They found that CART is amenable to achieving a preferred balance between the two types of classification rates (false-negatives and false-positives). Moreover, they concluded that process and execution metrics could also be significant predictors.

Gyimothy, Ferenc, and Siket [104] demonstrated how object-oriented metrics proposed by Chidamber and Kemerer [51] could be used to fault-proneness detection of the source code of the open source Web and e-mail suite called Mozilla. They employed statistical (logical and linear regression) and machine learning (decision trees and neural network) methods and compared the values obtained against the number of bugs found in called Bugzilla, a bug database, to validate the usefulness of these metrics for fault-proneness prediction. They found concluded Coupling Between Object (CBO) metric seems to be the best in predicting the fault-proneness of classes. They also affirmed that the precision of their models was not satisfactory.

Singh, Kaur, and Malhotra [248] and Malhotra, Kaur, and Singh [170] applied Support Vector Machine (SVM) model to find the relationship between object-oriented metrics given by Chidamber and Kemerer [51] and fault proneness at different severity levels. Particularly, the authors investigated how accurately these metrics predict fault proneness when severity is taken into account. They concluded that SVM method predicts faulty classes with high accuracy. Mainly when applied to predict medium severity faults.

Shin and Williams [244] investigated whether execution complexity metrics can be used as indicators of vulnerable code locations to improve security inspection and testing efficiency. For this purpose, the authors performed empirical case studies on two widely used open source projects. They compared the effectiveness of execution complexity metrics and static complexity metrics in detecting vulnerable code locations using logistic regression. They concluded that models built using execution complexity metrics also could predict vulnerable code locations with similar prediction performance to the model using the combined set of complexity metrics, but with lower inspection effort.

Chowdhury and Zulkernine [52] investigated the efficacy of applying complexity, coupling, and cohesion metrics to predict vulnerability-prone entities in software systems automatically. They used different machine learning techniques (Decision Trees, Random Forests, Logistic Regression, and Naïve-Bayes) to build vulnerability predictors that learn from the complexity, coupling, and cohesion metrics and vulnerability history. The authors empirically evaluated their prediction model over the vulnerability history of more than four years and fifty-two releases of Mozilla Firefox. They found that their prediction models were able to correctly predict almost 75% of the vulnerability-prone files, with a false positive rate of below 30% and an overall prediction accuracy of about 74% independently of the learning method.

Shin and Williams [245] investigated whether fault prediction models can be used for vulnerability prediction or if specialized vulnerability prediction models should be developed when both models are built with traditional fault prediction metrics. To achieve this goal, they performed an empirical case study on a widely-used open source project. They built both fault prediction models and vulnerability prediction models using the three types of traditional fault prediction metrics and measured how accurately the models predict vulnerable code locations. The authors concluded that traditional fault prediction metrics could detect a high portion of vulnerable code locations but should be significantly improved to reduce false positives while providing high recall.

Walden, Stuckman, and Scandariato [271] compared the predictive power of vulnerability prediction models based on text mining with models using software metrics as predictors. For

this purpose, they built both models using a Random Forest machine learning technique and applied them to predict vulnerability in three web applications written in PHP. They found that text mining models had higher recall than software metrics based models for all three applications.

Lessmann et al. [154] empirically compared 22 classification models for software defect prediction over 10 public domain software development datasets. Comparing the individual models statistically, they found that most methods' predictive accuracy does not differ significantly. Therefore, they concluded that the assessment and selection of a classification model should not be based on predictive accuracy alone but should comprise several additional criteria like computational efficiency, ease of use, and especially comprehensibility

Zimmermann et al. [293] investigated the problem of cross-project defect prediction, i.e., computing a prediction model from a project and applying it to a different one. They studied 12 real-world applications for this purpose, and for each application, they collected code measures, domain and process metrics, and defects. Using these metrics, they built defect prediction models based on logistic regression. In total, they ran 622 cross-projects experiments. Their experiments showed that using models from projects in the same domain or with the same process does not lead to accurate predictions. Moreover, the authors identified important factors influencing the performance of cross-project predictors.

Al-Jamimi and Ghouti [126] evaluated fault prediction models based on support vector machines and probabilistic neural networks. They compared the performance of the two approaches concerning their prediction accuracy against each other in the context of the five different datasets. The results indicated that the probabilistic neural networks generally provided the best prediction performance for large datasets experiments.

Jiang, Tan, and Kim [130] explored the idea of building separate prediction models for individual developers to predict software defects. To validate this idea, they applied these personalized prediction models to change classification. They evaluated these models on six open source projects using different classification algorithms (AdTree, Naïve Bayes and Logistic regression). They found that their personalized change classification models outperform the traditional change classification and multivariate adaptive regression splines models independently of the classification algorithm.

Taba et al. [255] explored the possibility of predicting bugs using antipatterns to improve the accuracy of state-of-the-art bug prediction models. They proposed four metrics based on antipatterns' history in a file to capture antipatterns information in software systems. Using these metrics, the authors built logistic regression models to compare each new antipattern based metric to the state of the art models based on product and process metrics. They concluded that antipattern metrics provide additional explanatory power about the bug-proneness of files over product and process metrics.

Kreimer [148] introduced a method for detecting code smells combining object-oriented metrics with machine learning (Decision trees). They extract these metrics from Java byte-code. As a result of this initial study, he found that models built using his idea reached an accuracy of 95% and 100% when detecting 'Long Method' and 'Big Class' respectively.

Maneerat and Muenchaisri [173] presented a methodology for predicting bad smells from software design model. They applied different machine learning algorithms to detect 7 bad smells. They concluded that no machine learning algorithm could accurately predict all selected bad smells.

Maiga et al. [168, 167] introduced an approach to detect anti-patterns, based on a machine learning technique - support vector machines — that consider practitioners' feedback. Through an empirical study involving three systems and four anti-patterns, they showed that their approach has better accuracy than two state-of-the-art approaches, respectively, that apply exact

and probabilistic anti-patterns detection.

Fontana et al. [85] presented an experimental analysis of the application of machine learning to code smell detection. The authors described their approach that was implemented and evaluated using different classifier algorithms: Support Vector Machines, Decision Trees (J48), Random Forest, Naïve Bayes and JRip. They concluded that their results could be used as a baseline for a community effort in finding the best possible code smell predictor.

Arcelli Fontana et al. [18] compared 16 different machine learning algorithms to detect four code smells on 74 software systems. They found that all algorithms achieved high performances in the cross-validation data set, yet the highest performances were obtained by J48 and Random Forest, while support vector machines achieved the worst performance. Moreover, they concluded that the application of machine learning to the detection of these code smells can provide accuracy high than 96%.

Pecorelli et al. [220] conducted a large-scale empirical study to compare the performance of heuristic-based and machine learning-based techniques for metric-based code smell detection. The authors compared five code smell prediction models built using different algorithms with DECOR [195], a state-of-the-art heuristic-based approach. They evaluated machine learning techniques, and heuristic approaches considering five different types of code smells over a dataset composed of 125 releases of 13 open source software systems. They concluded that heuristic approaches still perform better yet have low performance.

Sahs and Khan [238] built an Android malware detector with features based on a combination of Android-specific permissions and a Control-Flow Graph representation. The authors evaluated their classifier using with k-Fold cross-validation on a dataset of 91 malware and 2081 benign applications. They concluded that although their classifier had a very low false-negative rate, there was also much room for improvement in its high false-positive rate.

Amos, Turner, and White [9] compared 6 machine learning classifiers on 1330 malicious and 408 benign applications, for a total of 1738 unique applications. They concluded that cross-validation results were not consistent with results from real testing and over-estimate performance.

Demme et al. [63] investigated the feasibility of building a malware detector in hardware using dynamic performance data. They applied standard machine learning classification algorithms such Decision Trees to detect variants of known malware. They concluded that classification algorithms could detect malware at nearly 90% accuracy with 3% false positives for some mobile malware.

Yerima et al. [285] proposed and evaluated a machine learning approach based on Bayesian classification for detecting Android malware. Their models were evaluated with real malware samples in the wild. The results demonstrated a significantly better detection rate compared to the achieved by popular signature-based antivirus software.

Canfora, Mercaldo, and Visaggio [41] proposed a method for detecting malware based on two classes of features: invoked system calls and permissions that the application under analysis requires. To evaluate five classification algorithms, they used a dataset composed of 200 trusted and 200 malware Android applications from different categories. They concluded that their results must be considered as preliminary findings for further studies.

Allix et al. [5] compared the performance of machine learning classifiers in the laboratory and 'in the wild'. They designed multiple machine learning classifiers and built a set of features that are textual representations of basic blocks extracted from the Control-Flow Graph of applications. They concluded that Android malware detectors had poor overall performance in the wild, unlike in the laboratory tests.

Fereidooni et al. [83] proposed a machine learning-based system to detect malicious Android applications through static analysis of Android applications. To this end, the authors employed

several algorithms for the classification and evaluated them using a data-set containing 18 677 malware and 11 187 benign applications. The results showed a true positive rate of 97.3% and a false negative rate of 2.7%.

Feng et al. [82] proposed a framework for malware detection on Android applications combining dynamic analysis with machine learning. Their framework extracts dynamic behavior features by monitoring operations of benign and malicious applications to train ensemble learning model and distinguish malicious from benign applications. To evaluated their framework, they compared its detection performance with a state-of-art dynamic analysis tool and concluded that their framework overcomes that tool.

Zhu et al. [292] proposed an ensemble machine learning system for the detection of malware on Android devices. Their system extracts four groups of features including permissions, monitoring system events, sensitive API and permission rate to characterize each Android application. Then an ensemble random forest classifier is learned. They evaluated their system's performance using a dataset composed of 2 130 Android applications, of which 1 065 are benign and 1065 are malware. The experimental results demonstrated their system achieved 89.91% accuracy, outperforming a state-of-the-art support vector machine classifier.

Cai et al. [38] developed a machine learning-based technique to detect and categorize Android malware. They trained their model using the Random Forest algorithm with features that cover run-time application characteristics. They evaluated this model against two state-of-the-art peer approaches as baselines on 34 343 distinct applications spanning 2009 through 2017. The authors concluded that their technique outperformed the baselines in stability, classification performance, and robustness, with competitive efficiency.

2.4.2 Learning-to-rank applied to software engineering

Learning-To-Rank (LTR) refers to machine learning techniques for training models in a ranking task. These models can be used to sort objects according to their degrees of relevance, preference, or importance, as defined in a specific application [159]. This technique has been applied to specific problems related to software maintenance and evolution, as we report in this section.

Xuan and Monperrus [279] presented a learning-based approach that combines multiple fault localization ranking metrics. The authors empirically their against seven ranking metrics and concluded that it could localize faults more effectively than the ranking metrics taken in isolation.

Ye, Bunescu, and Liu [283, 284] developed a learning-to-rank approach that emulates the bug-finding process employed by developers. They trained a ranking model that characterizes useful relationships between a bug report and source code files by leveraging domain knowledge. The authors empirically evaluated their approach and conclude that it outperforms the other three state-of-the-art approaches.

Zhao et al. [290] evaluated the approach created by Ye, Bunescu, and Liu [283] to verify the influence of the recommended files' size on the efficiency in detecting bugs. Comparing with the Standard Vector Space Model (VSM) approach and the Usual Suspects (US) approach on six large-scale Java projects, they concluded that the approach proposed by Ye, Bunescu, and Liu [283] performs worse than VSM in terms of code inspection effort to detect bugs.

Yang, Tang, and Yao [282] introduced a learning-to-rank approach to build software defect prediction models by directly optimizing the performance measure. They compared their LTR method against other algorithms that optimize the individual-based loss functions and concluded that their approach could give a better ranking than these algorithms.

Le et al. [151] proposed a fault localization approach that employs a learning-to-rank strategy, using likely invariant diffs and suspiciousness scores as features. They evaluated their approach

on 357 real-life bugs. They concluded that it could successfully locate more bugs than several state-of-the-art spectrum-based fault localization baselines.

Tian et al. [261] created a learning-to-rank model that combines location-based and activity-based information from historical bug reports to recommend developers automatically to address particular bug reports. They evaluated the proposed approach over more than 11 000 bug reports and concluded that their model performs the best when compared to state-of-the-art location-based and activity-based models.

Niu, Keivanloo, and Zou [203] proposed a code example search approach based on the learning-to-rank technique. They identified 12 features from 2 500 code examples for 50 queries and applied a learning-to-rank algorithm to learn how the 12 features should be combined to build a ranking schema. Their approach was evaluated using a corpus of over 360 000 code snippets crawled from 586 open source Android projects. Finally, the authors concluded that their approach outperforms Codota, a commercial online code example search engine for Android application development.

Wang, Huang, and Ma [272] presented a top-k learning-to-rank approach to Cross-Project Defect Prediction (CPDP). They proposed a new data resampling method. Then, they evaluated several learning-to-rank algorithms using the proposed method as the data resampling method and concluded that it outperforms other data resampling algorithms.

Cao et al. [42] proposed a rule-based specification mining approach based on learning-to-rank. Their approach takes known specification rules as input and learns the best combination over the 38 interestingness measures. They compared their approach with state-of-the-art approaches that considered single interestingness measure and concluded that the learning-to-rank approach outperforms the best by up to 66%.

Loyola, Gajananan, and Satoh [165] introduced a learning-to-rank based model to support bug localization. This model learns feature representations from source changes extracted from the project history and code change dependency. Comparing their model against the state-of-the-art bug localization solutions, the authors reported a competitive result.

Shi et al. [243] surveyed hybrid bug localization methods. Combining features from hybrid methods with different learning-to-rank techniques. In total, they compared eight learning-to-rank algorithms applied in bug localization. The results showed that the coordinate ascent algorithm without normalization is a suitable learning-to-rank method, outperforming two state-of-art approaches.

Kim et al. [143] presented a learning-to-rank fault localization technique that uses genetic programming to combine multiple sets of localization input features. Their approach combines static and dynamic features using genetic programming. The evaluation results showed that their approach outperforms the state-of-the-art mutation based fault localization, spectrum-based fault localization, and learning-to-rank fault localization techniques significantly.

Sohn and Yoo [249][250] introduced a learn-to-rank fault localization approach that learns how to rank program elements based on existing spectrum-based fault localization formulas, code metrics and change metrics. The performance of their approach was evaluated using 386 real-world faults.The authors concluded that their approach performs significantly better than the state-of-the-art spectrum-based fault localization formulas.

Bertolino et al. [26] compared two strategies for machine learning-based prioritization, learning-to-rank, and reinforcement learning, to prioritize tests in continuous integration. They evaluated the ten algorithms classified into ensemble and non-ensemble algorithms. As a result, testers have devised guidelines to select and tune the ML algorithms best fitting their needs.

Haas and Hummel [105] applied to learning-to-rank to derive a scoring function to suggest extract method refactoring of long Java methods. They empirically analyzed their candidate scoring function using a set of 177 long methods and a total of 1,185 refactoring candidates.

The authors concluded that the resulting scoring function needs fewer parameters than other state-of-art scoring functions, but it has a better ranking performance.

Hussain et al. [122] used an LTR method to create a prototype of an automated recommendation system to classify and select design patterns. Given a design problem description, their approach ranks the design patterns according to the text relevancy. They evaluated the efficacy of the proposed method in the context of several design pattern collections and relevant design problems. The authors concluded that their prototype is capable of selecting the right design pattern(s).

2.4.3 Summary

In this section, we reviewed studies that applied classification and learning-to-rank to solve different software engineering problems. The majority of these classification studies focused on maintainability prediction, defect prediction, code smell detection, and fault-proneness prediction. On the other hand, most of the learning-to-rank studies focused on fault localization and defect prediction. Additionally, we described some studies in which learning-to-rank was used to create recommendation systems, for instance, recommending methods refactoring or design patterns to be applied. However, we concluded that classification and learning-to-rank algorithms had not been employed to recommend code migration.

2.5 Conclusion

Based on our literature review, we observe that the research community still lack knowledge about various aspects related to the usage of Kotlin in the context of the development of Android applications:

- **Adoption of Kotlin:** When we started this research in January 2018, there were no studies about the adoption of Kotlin. Therefore, it was necessary to investigate whether Android developers were adopting Kotlin and its impact on Android applications' quality. Since we started working on this topic, some studies have covered similar research topics. For example, Flauzino et al. [84] compared Android applications written in Java and Kotlin, Coppola, Ardito, and Torchiano [59] investigated the adoption of Kotlin and how it impacts Android application's popularity, and Oliveira, Teixeira, and Ebert [205] examined the perception of developers about the advantages and disadvantages of adopting Kotlin. Thus, there is a research opportunity for understanding how the adoption of Kotlin impacts application source-code and, consequently, on Android applications quality. Besides, the potential findings of this research could lead developers to write better Android applications.
- **The usage of Kotlin:** Despite the papers about adopting Kotlin by Android [59], no paper investigated how Kotlin and its features have been used to create Android applications. As Kotlin brings features not available in Java, we want to know whether Android developers are using them. Moreover, we believe that understanding how developers are coding Android applications with Kotlin is essential to improve these applications' quality. Consequently, this knowledge could be used to assist developers in migrating their applications from Java to Kotlin.
- **Migration to Kotlin:** The announcement that Android has become 'Kotlin-first' revealed Google's intention, the owner of Android, of turning the Android ecosystem to Kotlin. Furthermore, after this announcement, Kotlin became Google's recommended choice for

mobile application development. Considering this scenario, companies that want to keep updated with the most recent Android features would need to use Kotlin. Consequently, we believe that companies and developers will have an interest in migrating their applications to Kotlin. However, there is no research about the migration of Java code to Kotlin.

In this thesis, we address these knowledge gaps and provide necessary elements to understand how Kotlin impacted on Android application development, how it has been used, and finally, how we could assist the migration of Android applications to Kotlin. We start by addressing the lack of studies about the adoption of Kotlin. In the upcoming chapter, we present an empirical study that explores the adoption of Kotlin and its impact on the quality of Android applications.

Chapter 3

Measuring the adoption of Kotlin by Android developers

During the last years, different development approaches and frameworks have emerged to ease mobile applications' development. [200, 180]. Meanwhile, Google has continued evolving its development toolkit for building applications. For instance, Android 7.0 added MultiWindow support [66], Android 8.0 redesigned notifications to provide an easier and more consistent way to manage notification behavior and settings [67] and Android 10 included extensive changes to protect privacy and give users more control [65]. Following the same trend adopted by its competitor, Apple, that announced in 2014 a new programming language (Swift) for the development of mobile applications, Google announced in 2017 Kotlin as an official Android programming language. However, as showed by the state of the art, the impact of this announcement in the adoption of Kotlin was not known, since there were no studies about the use of Kotlin by Android developers.

Kotlin is 100% interoperable with the Java programming language, so it is possible to call Java-based code from Kotlin, or to call Kotlin from Java-based code (See Section 2.1). Consequently, developers may create applications using Kotlin in different ways: for instance, writing new applications using only Kotlin or adding new features in applications written initially in Java, or even migrating their applications.

For these reasons, this chapter aims to study the adoption of Kotlin on Android applications. More specifically, we conducted an empirical study to answer the following research questions:

- RQ_1**:** What is the degree of adoption of Kotlin in Android open source applications?
- RQ_2**:** What is the proportion of Kotlin code in mobile applications?

To carry out our study, first, we collected more than 2 000 open source Android applications that have at least one version released in the same year that Kotlin became an Android official programming language or later. Then, we mined these applications to compute the amount of Kotlin code on each application. Applying this methodology, we identified Android applications based on Kotlin that are targets of other empirical studies presented along this thesis.

The chapter continues as follows. Section 3.1 presents the steps needed to create the dataset of Android applications written in Kotlin used in this thesis and the empirical study to measure the adoption of Kotlin by Android developers. Section 3.2 describes the empirical study's study design to investigate the amount of Kotlin code in Android applications and its results. Section

3.3 presents threats to validity. Finally, Section 3.4 summarizes our work and outlines further works to perform on this research topic.

The content of this chapter has been published in the Empirical Software Engineering journal [99].

3.1 Study of the adoption of Kotlin

This section focuses on our first research question:

- RQ_1**:** What is the degree of adoption of Kotlin in Android open source applications?

Initially, to answer this research question, we collected as many as possible open source Android applications. Section 3.1.1 describes the criteria and steps used to create this dataset. Section 3.1.2 presents the heuristics for classifying these Android applications into Kotlin applications and Java applications. Finally, Section 3.1.3 presents and discusses the results of the empirical study conducted to answer RQ_1.

3.1.1 Looking for Kotlin-based Android applications

To identify Android applications using Kotlin, we focused on Android applications with at least one version released in 2017, when Kotlin became an official language for Android development or later. We considered that applications whose last versions date from 2016 or earlier could not give us much information about the use of Kotlin language in the Android domain.

To carry on our study, we need for each application: *a)* its source code hosted in a code repository (e.g., GitHub), and *b)* binary files (apk) of the released versions. For those reasons, we mined three defined datasets of Android applications: 1) F-droid[1], which is the base of several datasets of Android applications found in the literature [95], 2) AndroidTimeMachine [96], a dataset that does not rely on F-droid, and provides source code of applications published on Google Play, consequently, minimize the probability of an application be a toy project, and AndroZoo [6, 161], the largest dataset of Android executable, from where we only extracted binary files of applications found either F-droid or AndroidTimeMachine. Let us introduce each dataset and explain the reasons for choosing them.

F-droid is a directory of open source Android applications that contains 1 509 applications.[2] On the main page of each application, F-Droid provides links to download the last three versions of an application and a link for another page that contains a list of all application versions. F-droid provides all the information we need, i.e., access to a code repository and the apks of each released version of an application. However, the number of applications (1 509) is relatively small compared with other datasets. For this reason, we decided to mine other Android datasets to include more applications in our study.

AndroidTimeMachine is a graph database of Android applications which are both accessible on GitHub and Google Play [96]. To create this dataset, the authors defined and executed a 4-step process: *1)* in the first, are identified open source Android applications hosted on GitHub, *2)* in the second step, are extracted their package names, *3)* in third step, are checked their availability on the Google Play store, and *4)* in the fourth step, are matched each GitHub repository to its

[1] https://f-droid.org
[2] Last visit: 06/04/2018.

corresponding app entry in the Google Play store [96]. In total, AndroidTimeMachine has 8,431 applications, and it is based on a publicly-available GitHub mirror available in BigQuery.[3] Using the Neo4j database, it is possible to retrieve the source-code repository link and the application's package name for each application. However, this dataset does not provide any apks from its applications. For this reason, we decided to mine the missing apks on the AndroZoo dataset.

AndroZoo is a dataset of millions of Android applications collected from various data sources [6, 161], including major market places *Google Play*, *Anzhi*, and *AppChine*, as well as smaller directories mobile, *AnGeeks*, *Slideme*, *ProAndroid*, *HiApk*, and *F-Droid*. In total, AndroZoo has 4 390 288 applications corresponding to a total of 7 795 372 apks (versions).[4] The list of available apks is regularly updated on AndroZoo website, along with metadata for each application as the main package name, the size of the apk, the version, and the market where the app was downloaded from, etc. [6, 161] However, AndroZoo does not provide any information about the source-code repository of an application, even if it is open source.

Now, we detail the steps executed to build the dataset used in this study. Figure 3.1 illustrated these steps.

Step 1 - Mining F-Droid: Using the date of upload of the application's version added in F-Droid, we retrieved 926 applications from F-Droid that fulfill our selection criterion.[5] The total number of versions (i.e., apks) found corresponding to those applications was 13 094. We could download 11 675 apks (89%), because 11% of apks were not available.

Step 2 - Mining AndroidTimeMachine: The version of AndroidTimeMachine presented by Geiger et al. [96] contains 8 431 applications and it was executed in October 2017. As their infrastructure is publicly available,[6] we re-executed it in October 2018 to include more recent applications.

Once we retrieved the list of applications, differently of the original work from Geiger et al. [96], we carried out a new step to keep applications whose code repositories have only one Android manifest file. We added this new step because we discovered that the AndroidTimeMachine match algorithm produced wrong results (i.e., applications linked with wrong repositories) when a repository has more than one *AndroidManifest.xml*. This list of applications that suffer from the mentioned problem is publicly available [183].

Table 3.1 shows the comparison between the numbers found by us resulting from the re-execution of each step and the numbers of the original dataset by Geiger et al. [96]. In conclusion, we retained 2 156 applications from AndroidTimeMachine.

Step 3 - Combining AndrodTimeMachine with AndroZoo: Since AndroidTimeMachine does not provide apks from the applications, we retrieved from AndroZoo the apks corresponding to each application collected from AndroidTimeMachine. To obtain those apks, we first extracted the package name located on the *AndroidManifest.xml*. Then we queried the AndroZoo HTTP API. In total, 1 531 applications from AndrodTimeMachine were found in AndroZoo. To ensure that these applications were correctly linked with repositories, we manually checked, for each

[3] `https://cloud.google.com/bigquery/public-data/github`

[4] Last visit: 16/10/2018.

[5] Executed on June 4th, 2018.

[6] AndroidTimeMachine resources: `https://androidtimemachine.github.io/dataset/` and `https://github.com/AndroidTimeMachine/open_source_android_apps`

Table 3.1: Steps executed to build a recent version of AndroidTimeMachine.

Steps	AndroidTimeMachine	
	Original	Updated
Finding Android Manifest Files	378 610	358 518
Extracting package Names	112 153	114 152
Discarding repositories with more than one manifest file	-	49 570
Filtering applications on Google Play	9 478	3 664
Matching of Google Play pages to GitHub repositories	8 431	3 664
Filtering applications from 2017 or later	-	2 156

application, its repository page and its page on Google Play Store. We removed 54 applications linked with wrong repositories, ending with 1 477 applications.[7]

Step 4 - Combining F-Droid and AndroZoo: Finally, we kept 1 241 applications from AndroZoo. The rest, 236, were already discovered from F-Droid during step 1.

The resulting collection: Our final collection corresponds to:

$$Fdroid \cup (AndroZoo \cap AndroidTimeMachine)$$

In total, we collected 2 167 applications (926 + 1 241) and 19 838 apks. Figure 3.2 shows the distribution of apks (released versions) per application. From F-droid we downloaded in median 6 apks. Moreover, from AndroidZoo, we downloaded in media 3 apks per application. In summary, we have in median 4 apks per application. The list of collected applications is publicly available in our appendix.[8]

3.1.2 Analysis method

To measure the adoption of Kotlin, we built a process to classify both applications and apks of our dataset in three categories of applications: *1)* applications written with Java (i.e., applications that do not include any line of Kotlin code), *2)* applications written partially with Kotlin, and *3)* applications written totally with Kotlin, we call these applications as 'pure Kotlin'. Note that we only focused on applications' source code and bytecode, discarding third-party libraries, which neither their source code nor bytecode (jars) are included in the applications' code repositories.

For each collected application, which has a link to the code repository and a set of apks, we applied three heuristics to classify an application and its apk. Figure 3.3 shows this classification process.

We first applied the heuristic H_{apk}, Figure 3.3(a), which consists of looking for a folder called **kotlin** inside the apk file. Having that folder indicates the presence of Kotlin code. To automatize this task, we used a tool named *apkanalyzer* included in the Android SDK. Using this heuristic, we first classified each version (apk) of an application. If at least one apk is classified as Kotlin,

[7] In the appendix we list all applications that suffer the mentioned problem: `https://github.com/UPHF/kotlinandroid/blob/master/docs/wrong_match.md`.

[8] FAMAZOA dataset: `https://github.com/UPHF/kotlinandroid/blob/master/docs/final_dataset.md`

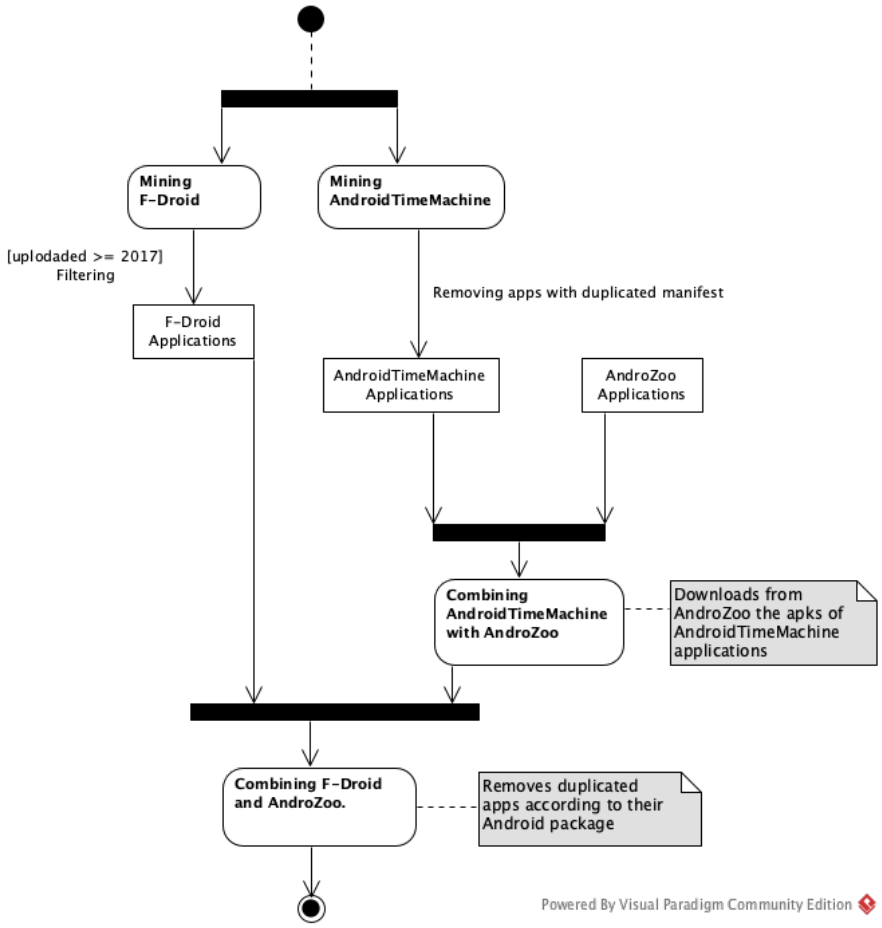


Figure 3.1: The steps needed to create the dataset target of this study.

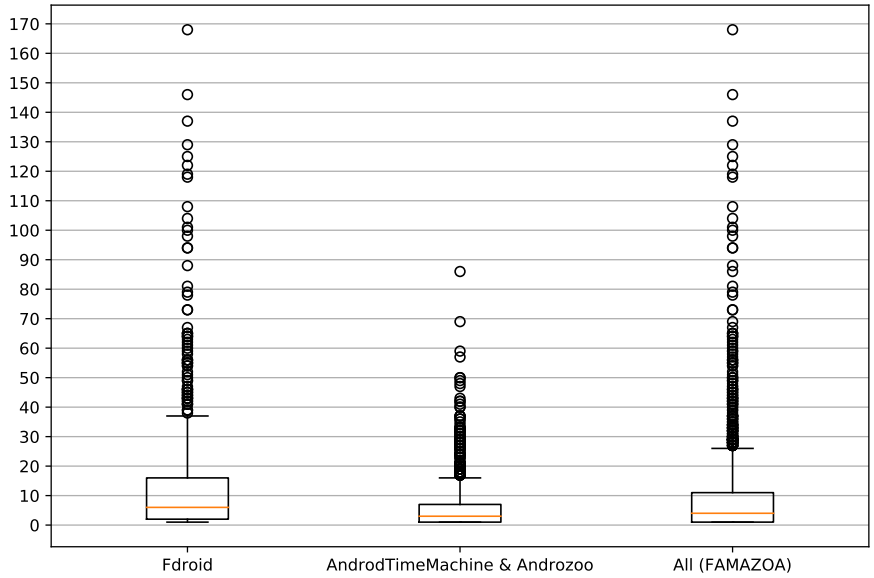


Figure 3.2: Distribution of number of versions (apk) per application.

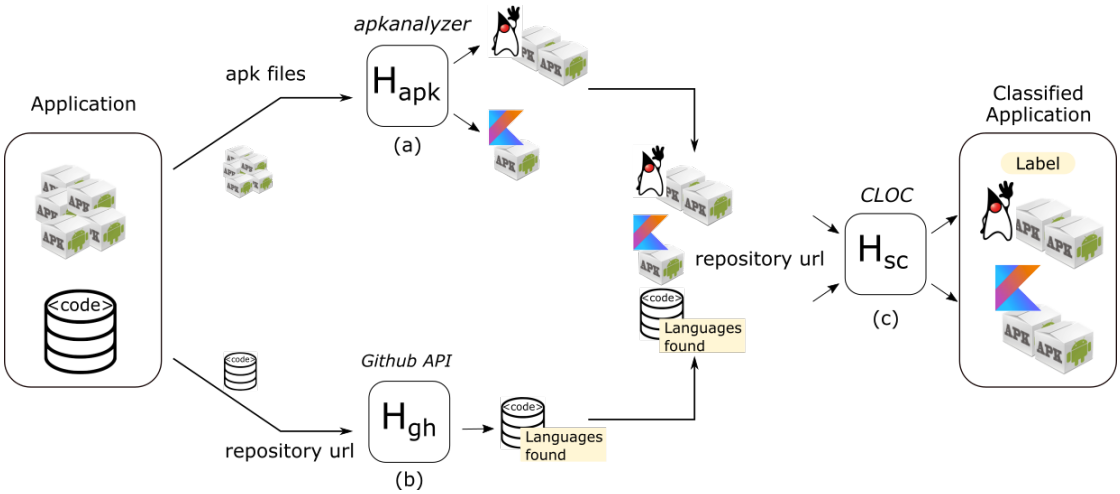


Figure 3.3: Our pipeline to classify Android applications (Section 3.1.2.).

Table 3.2: Classification of applications according to their programming language.

Information	Total	Kotlin	Java
Unique apps	2 167	244	1923
Versions	19 838	1590	18248

the heuristic classifies the application as 'Kotlin'. Otherwise, it classifies as 'Java'. The H_{apk} provides a cheap and fast approach to get an initial guess about the presence of Kotlin code.

At the same time, we applied our second heuristic H_{gh}, Figure 3.3(b), which relies on the GitHub API. For each application hosted on GitHub, we queried the GitHub API to retrieve the amount of code (expressed in bytes) from the most recent version grouped by programming language. We classified the app as 'Kotlin' if Kotlin is found in the API's response.

Finally, once we retrieved a set of candidate Kotlin applications using H_{apk} and H_{gh}, i.e., $H_{apk} \cup H_{gh}$, we applied the heuristic H_{sc} over them, Figure 3.3(c), to assert the presence of Kotlin code and to measure how much Kotlin an application has. The heuristic H_{sc} inspects every commit of a given application's repository. For this purpose, the heuristic used CLOC[9], a tool that counts lines of code in many programming languages,which returns a list with the programming languages used in an application and the amount of code not considering blank lines. H_{sc} is time-consuming because it requires analyzing the source code of each commit of a repository. Therefore, to execute H_{sc} for every application from our dataset is prohibitive.

Different from H_{sc}, heuristic H_{gh} only focuses on the most recent application version hosted on GitHub (the API only retrieves that information). Consequently, it cannot detect applications that: *a)* do not contain Kotlin code in the most recent version (last commit), but *b)* contain Kotlin code in older versions.

At the end of the classification process, an application is classified as 'Kotlin' if at least one commit that has Kotlin code is found and as 'Pure Kotlin' if all commits contained only Kotlin code.

3.1.3 Results

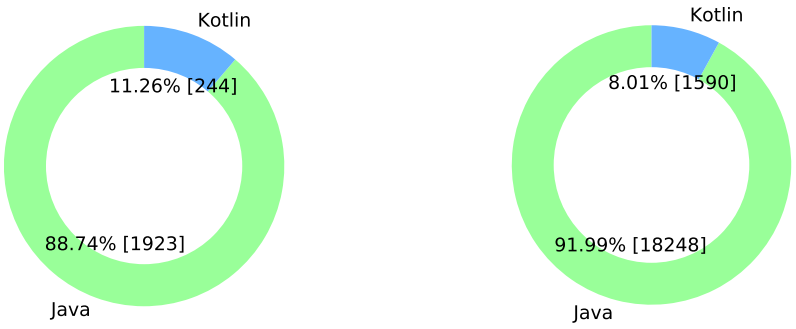
Table 3.2 summarizes the classification of applications done using the presented methodology. Our collection has 2 167 applications and using our heuristics we classified 244 (11.26%) as 'Kotlin' applications. Consequently, the remaining applications, 1 923 (88.74%), were classified as 'Java'. Figure 3.4a shows these percentages. Considering the number of versions (apk), we found 1 590 apks (8.01%) with Kotlin code and 18 248 (91.98%) without Kotlin code.

Now, let us explain how we arrived to detect 244 Kotlin applications. First, the heuristic H_{apk} (apk analyzer, Section 3.1.2) classified 265 applications as 'Kotlin'. Those applications have, at least, one apk classified as 'Kotlin'. For 76 of them, all apks are classified as 'Kotlin'. Then, H_{gh} (GitHub API, Section 3.1.2) classified 234 applications as 'Kotlin'. 193 of them were also classified as 'Kotlin' by H_{apk}. Up to here, both heuristics have classified 297 unique applications as 'Kotlin'.

Finally, we applied H_{sc}(Source Code analysis Section 3.1.2) on those 297 applications' repositories, finding 244 that contains Kotlin code. The list of Kotlin applications can be found in our appendix.[10]

[9] `http://cloc.sourceforge.net/`

[10] Applications classified as Kotlin: `https://github.com/UPHF/kotlinandroid/blob/master/docs/final_kotlin_dataset.md`



(a) Unique applications.

(b) All versions (apk).

Figure 3.4: Distributions between Kotlin and Java applications and versions.

We named this set of Android applications totally or partially written in Kotlin as FAMAZOA (F-droid AndroidtimeMachine AndroZoo open source Applications) [185]. FAMAZOA is, to the best of our knowledge, the largest publicly available dataset of Kotlin-based Android applications. Along this thesis, we conducted empirical studies (Chapters 4, 5 and 6) that target FAMAZOA's applications.

Response to RQ_1**:** *What is the degree of adoption of Kotlin in Android open source applications?*
We found that 244 out of 2 167 (11.26%) applications from our dataset have, at least, one version released between the years 2017 and 2018 written (totally or partially) using the Kotlin language.

This result indicates that Android developers are using Kotlin, although there is no scientific evidence that this choice positively impacts the development of applications. Therefore, this finding opens research opportunities. This thesis focus on some of those opportunities. In Chapter 4, we compare the occurrences of code smell in Kotlin-based and Java-based applications and based on that, we analyze the impact of Kotlin on the quality of Android applications. In Chapter 5, we analyze how Kotlin code is evolving in terms of lines of code and the use of Kotlin features along the development of Android applications.

3.2 Study of The proportion of Kotlin code in Android applications

Thanks to the interoperability between Java and Kotlin, Android developers can decide between writing applications totally or partially (with different proportions) in Kotlin. For instance, one application may have all its codebase written in Kotlin, whereas another one may have only 10% of lines of code written in Kotlin. For that reason, in this section, we analyze the amount of Kotlin code in Android applications by answering our second research question:

- RQ_2**:** What is the proportion of Kotlin code in mobile applications?

To answer this research question, we mined Kotlin applications from FAMAZOA. Figure 3.5 illustrates the steps followed to conduct this study. Section 3.2.1 explains the reason why we

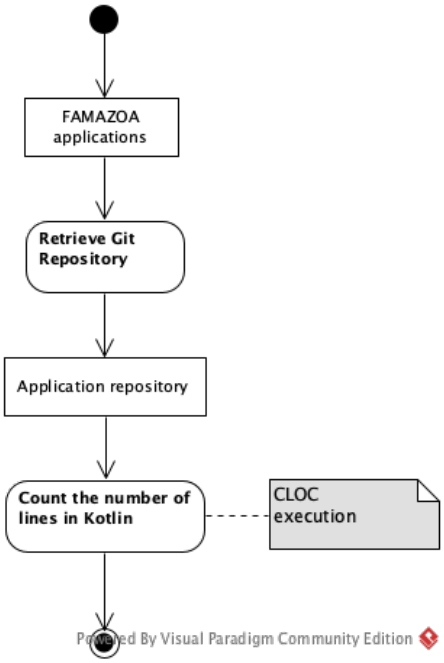


Figure 3.5: To answer our RQ_2, we retrieved the corresponding code repository (e.g., a git repository) of each FAMAZOA's application. Then, for each repository, we executed CLOC to identify the proportion of Kotlin code.

target these applications and characterizes them. Section 3.2.2 presents the method applied to the target applications to identify their Kotlin and Java code proportions. Finally, Section 3.2.3 presents and discusses the results of the empirical study conducted to answer RQ_2.

3.2.1 Applications analyzed in the study

The goal of this research question is to identify the proportion of Kotlin code in Android applications. For that reason, we did not mine all applications that we collected. We focused only in applications from FAMAZOA (244 applications).

3.2.2 Analysis method

For each application, we retrieved the corresponding code repository (e.g., a git repository). For each repository (associated with one application), we executed CLOC, a tool that counts lines of code in many programming languages, over the most recent version (i.e., the last commit), then we calculated the proportion of Kotlin code (excluding blanks and comments) concerning the total code (See Equation 3.1). In this analysis, we discarded files that did not contain Java or Kotlin code, such as XML, CSS, JavaScript, and others.

$$Proportion = \frac{Kt_{loc}}{LoC - BlackLines - Comments} \tag{3.1}$$

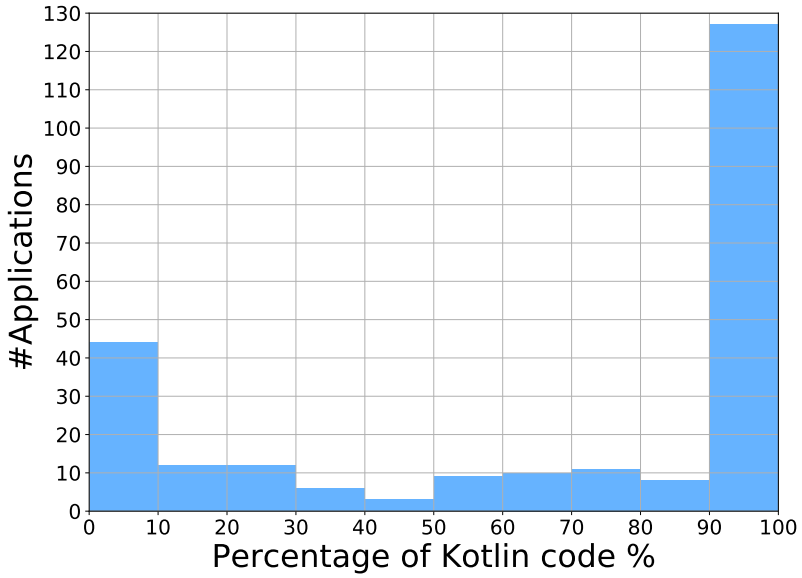


Figure 3.6: Distribution of applications according the percentage of Kotlin code. 59.43% of Kotlin applications have more than 80% of source code written in Kotlin.

3.2.3 Results

Figure 3.6 shows the distribution of Kotlin applications according to the percentage of Kotlin code. We found that 82 out of 244 (33.61%) applications have only Kotlin code. The rest of the Kotlin applications (66.39%) are also written in Java. Furthermore, we found that 145 out of 244 (59.43%) applications have at least 80% of Kotlin code and, on the contrary, 45 out of 2167 (18.44%) applications, have less than 10% of Kotlin code.

> **Response to** RQ_2**:** *What is the proportion of Kotlin code in mobile applications?*
> Considering the last version of each application, most Kotlin applications (59.43%) have at least 80% of lines of code written in Kotlin.

This result indicates that although 11% of Android applications adopt Kotlin, most of these applications are almost entirely (80% of its codebase) written in Kotlin. However, we do not know if developers are writing new applications or converting applications written in Java, or adding Kotlin code to applications initially developed in Java. We investigate this aspect in Chapter 4. Furthermore, Kotlin brings functional features together with object-oriented features to the development of Android applications. For that reason, Chapter 5 presents an empirical study focused on the use of Kotlin features by Android developers.

3.3 Threats to Validity

In this section, we discuss the main issues that may have threatened the validity of the studies presented in this chapter, considering the classification of threats proposed by Cook, Campbell,

and Day [58]:

3.3.1 Internal

Classification of Android applications. In Section 3.1 we defined a procedure for classifying Android applications in 'Kotlin' and 'Java' based on three heuristics that inspect both source code and Android binary files (apks). By applying those heuristics, we assure the absence of false-positives in our dataset, i.e., applications classified as 'Kotlin' but without Kotlin code along with their life-cycle. However, it could exist some false negatives, i.e., applications that: *1)* has versions with Kotlin, but heuristic H_{apk} does not classify them as Kotlin, and *2)* the last version from the code repository does not have Kotlin code anymore (by definition, not detected by the heuristic H_{gh}). The heuristic H_{sc} could detect such applications in case that H_{apk} fails. However, it is expensive (time-consuming) to execute H_{sc} over the complete set of collected applications considering our current infrastructure (H_{sc} analyzes each commit from the application's source code).

3.3.2 External

Representativeness of the set applications collected. Our studies focus on studying both source and byte code of mobile applications. For that reason, we decided to study the open source application available on GitHub. To our knowledge combining F-Droid, AndroidTimeMachine, and AndroZoo, makes our dataset the largest Android repository that has both binary and source code of each app. However, we cannot generalize our findings over applications that are not open source.

3.4 Summary

In this chapter, we presented an empirical study that explores the adoption of Kotlin by Android developers. Initially, we collected open source Android applications to investigate how many applications have been written using Kotlin. Once we found these applications, we measured the amount of Kotlin on it.

To perform our empirical study, we mined 3 existing Android datasets (F-droid, AndroidTimeMachine [96] and AndroZoo [6, 161]). In total, we collected 2 167 applications and 19 838 apks. To identify applications that have Kotlin code, we defined and applied 3 heuristics and we found that 244 out of 2 167 (11%) applications that have at least one released version between 2017 and 2018, adopted Kotlin. The result of this empirical study showed that around 11% of Android applications already adopted Kotlin. Then, considering the applications that have adopted Kotlin, we compared the amount of Kotlin and Java code in their last version (i.e., commit). The comparison showed that 145 out of 244 (59.43%) applications have at least 80% of their codebase written in Kotlin.

Regarding our thesis plan, this chapter allowed us to fill one particular gap: the absence of knowledge in the literature about the adoption of Kotlin by Android developers. Considering our result, we can affirm that Android developers have adopted Kotlin. However, at this point, we still do not know how the adoption of Kotlin impacts the quality of Android applications. In Chapter 4, we address this gap identified in our literature review.

Chapter 4

Measuring and Comparing the quality of Android applications written in Kotlin

As showed in Chapter 3, in our study conducted in 2018, we found that around 11% of Android applications have adopted Kotlin and among these applications, 66.39% mix Kotlin and Java. However, combining multiple programming languages in the same software increases its defect proneness [144]. Moreover, it is known that language design affects software quality [230]. Nevertheless, as shown in our review of state of the art (See Section 2.1, no study considers how the adoption of Kotlin impacts the quality of Android applications. For that reason, this chapter aims to investigate the impact of adopting Kotlin on Android applications' quality.

As proposed by Hecht et al. [115], we measure the quality of an Android application regarding the presence of code smells [87]. More specifically, we conduct an empirical study with the aim of answering the following research questions:

- RQ_3**:** Is there a difference between the quality of Kotlin and Java Android applications, expressed in terms of code smells presence?
- RQ_4**:** How frequent does the introduction of Kotlin positively impact on the quality of the versions of an Android application?

To carry out this study, we mined more than 2 000 Android applications to detect instances of ten code smells. We replicated the empirical study done by Habchi et al. [108], which compares iOS and Android applications to compare code smells found in two sets of Android applications: Android applications with Kotlin code and Android applications only written in Java. Moreover, we used the quality model proposed by Hecht et al. [115] to measure the impact on the quality of applications initially written in Java that have introduced Kotlin.

The chapter continues as follows. Section 4.1 presents the tool used to identify code smells in applications and the set of code smells considered in both studies described in Section 4.2 and 4.3. Section 4.2 describes our empirical study conducted to compare the presence of code smells on Kotlin and Java Android applications. Section 4.3 describes our empirical study conducted to investigate the impact of the adoption of Kotlin code on the quality of Android applications.

Section 4.4 reports threats to validity. Finally, Section 4.5 summarizes our work and outlines further works to perform on this research topic.

The content of this chapter has been published in the Empirical Software Engineering journal [99].

4.1 Study design

We present in this section the tool used in our empirical studies and the reasons for choosing it. Moreover, this section describes the code smells considered to answer both research questions.

We would like to recall that Kotlin and Java run on top of the Java Virtual Machine (JVM). Consequently, both languages share the same JVM bytecode. This property makes it possible to interoperate between both languages, allowing developers to call Java code from Kotlin code and vice versa. Moreover, as both languages share the same bytecode, we can use it to compare Java-based and Kotlin-based Android applications.

4.1.1 Tool selection

To answer our research questions, we need a tool capable of identifying code smells on Kotlin-based and Java-based Android applications. Moreover, due to the relevance of domain-specific code smells [14], we decided to analyze the presence of object-oriented and Android-specific code smells. Therefore, we looked for a tool capable of identifying both types of code smells on Kotlin-based and Java-based Android applications. Since developers use official/commercial markets to distribute their apks, most Java source code and bytecode existing analyzers could not be used to analyze the files [160].

Considering these requirements, we selected Paprika [115, 117]. Paprika is a static analysis tool that fits these criteria: *a)* it can detect object-oriented and Android-specific code smells [117]; *b)* it was designed for detecting code smells on Android applications without requiring the source-code: the input of Paprika is the apk of one Android application; *c)* as it works at JVM bytecode level (i.e., an apk contains bytecode), it can analyze Android applications written in Java and/or Kotlin. Moreover, Paprika has been extensively used for analyzing mobile applications [115, 117, 116, 108, 43, 101] and its implementation was deeply validated, including an experiment done together with Android developers [114].

Paprika can identify 4 object-oriented, and 13 Android code smells. Table 4.1 shows these code smells and it also presents the entities that are related to each of them.[1]

The input of Paprika is an apk (i.e., a version of an Android application). To detect occurrences of code smells, Paprika uses metrics associated with entities. For example, the code smell *Long Method* (LM) is an object-oriented smell, and it is related to *methods*: an instance of LM is a method in which the number of instructions is higher than a given threshold.

As output, Paprika produces *a)* a list of smells found, and *b)* the metrics associated with the entities used for detecting smells, for example, the number of methods, activities and services. Moreover, for each code smell's instance (incl. BLOB, CC, and HSS), Paprika outputs a fuzzy value (between 0 and 1) calculated using fuzzy logic [287], representing the degree of truth of the detected instance [108, 114].

[1] Version of Paprika used: commit 5ebd34 https://github.com/GeoffreyHecht/paprika/commit/5ebd349ed3067914386e8c6a05e87ff161f9edd1

Table 4.1: Paprika supported code-smells. The column 'Considered' shows the 10 code smells studied in our work (✓) and the 7 not studied (×).

Type	Code smell name	Entity	Considered
Object-Oriented	Blob Class (BLOB)	Class	✓
	Swiss Army Knife (SAK)	Interface	✓
	Complex Class (CC)	Class	✓
	Long Method (LM)	Method	✓
Android-Specific	Hashmap Usage (HMU)	Class	×
	Unsupported Hardware Acceleration (UHA)	Class	×
	Leaking Inner Class (LIC)	Inner class	×
	Member Ignoring Method (MIM)	Method	×
	Internal Getter/Setter (IGS)	Method	×
	No Low Memory Resolver (NLMR)	Activity	✓
	Heavy ASynctask (HAS)	Async Task	✓
	Heavy Service Start (HSS)	Service	✓
	Heavy Broadcast Receiver (HBR)	Broadcast Receiver	✓
	Init OnDraw (IOD)	View	✓
	Invalidate Without Rect (IWR)	View	×
	UI Overdraw (UIO)	View	✓
	Bitmap Format Usage (BFU)	-	×

4.1.2 Code smell selection

In this subsection we introduce the code smells considered in our empirical studies and the ones discarded.

4.1.2.1 Code Smells considered in our study

We now describe the code smells that we study in this thesis. Table 4.1 shows them with a ✓in column "Considered".

Firstly, we considered the four object-oriented smells (BLOB, SAK, and CC, related to classes, LM related to methods) because they can also exist in Kotlin applications. Let us briefly describe each of them. A Blob class (BLOB), also known as God class, is a class with many attributes and/or methods [32]. A Swiss army knife (SAK) is an interface with many methods [114]. A complex class (CC) is a class containing complex methods. These classes are hard to understand and maintain and need to be refactored [87]. On Paprika, the class complexity is calculated by summing the complexities of the internal methods and the complexity of a method is calculated using McCabe's Cyclomatic Complexity [189, 115]. Long methods (LM) have much more lines than other methods, becoming complex, hard to understand and maintain.

Secondly, we considered 6 Android platform related code smells that Paprika can detect, and we discarded 7. Those Android smells we considered are: *1)* NLMR (related to activities), *2)* HAS (async taks), *3)* HSS (async taks), *4)* HBR (broadcast receivers), *5)* UIO (views), and *6)* IOD (views)

Let us briefly describe each of them. *No Low Memory Resolver* (NLMR) [117] occurs when activities do not have the method *onLowMemory()* overridden. If an activity does not implement this method, the Android system could kill a process related to this activity to free memory. Consequently, it could cause an abnormal termination of programs [233].

Heavy ASynctask (HAS) [114], *Heavy Service Start* (HSS) [114] and *Heavy BroadcastReceiver* (HBR) [115] are similar: they occur when heavy operations are executed at the main thread in different Android components, Async Task, Service and BroadcastReceiver, respectively [178, 177, 176].

UI Overdraw (UIO) [115] and *Init OnDraw* (IOD) [114] are related to custom views. The smell UIO produces overdraw views because of missing methods invocations, such as *clipRect* and *quickReject* [188], which could avoid the overdraw. IOD happens when new objects are created inside the *onDraw* method that could be executed many times by second, resulting in many allocations of new objects [155].

4.1.2.2 Code Smells ignored in our study

We ignored 7 Android-related code smells that Paprika can identify. Table 4.1 shows them with a "×" in column "Considered". These smells are MIM, LIC, IGS, BUF, HMU, UHA and IWR. In the remainder of this section, we describe them and explain why we exclude them.

Member Ignoring Method (MIM) [233] occurs when a method does not access any class's attribute. In Android, it is recommended to use a static method instead because static method invocations are about 15%-20% faster than a dynamic invocation [11]. The smell *Leaking Inner Class* (LIC) [117] occurs when an application uses a non-static and anonymous inner class, since in Java, this type of inner class holds a reference to the outer class, and consequently, it could provoke a memory leak in Android systems [233, 164]. We decided to discard MIM and LIC because Kotlin does not have static methods [146].

Internal Getter/Setter (IGS) [233] impacts on performance and energy consumption of applications [116, 196, 197, 140, 101, 214, 43]. However, this code smell only impacts when an application runs on Android platforms 2.3 or less [62]. We discarded this smell because the number of active Android devices that run those platform versions is smaller than 0.5%.[2]

Bitmap Format Usage (BFU) is related to image format [43]. We discard it because it is related to neither Kotlin nor Java code, i.e., the smell is independent of the programming language used.

HashMap Usage (HMU) [43] occurs when developers use small **HashMap** instances instead of using **ArrayMap** or **SimpleArrayMap**, both provided by the Android framework [106, 10]. However, the results found by Saborido et al. [237] show that **ArrayMap** is generally slower and less efficient regarding energy consumption than **HashMap**. Moreover, they showed that when the keys used are primitive types, developers should adopt **SparseArray** variants because they are more efficient concerning CPU time, memory and energy consumption. We discarded this smell because of: *a)* the mentioned finding from Saborido et al. [237], and *b)* the Paprika's mechanism used to identify HMU occurrence does not take into account the key's type.

Finally, we discarded 2 smells related to custom views. *Unsupported Hardware Acceleration* (UHA) [114] occurs when developers call a method that is not hardware accelerated, so it runs on the CPU instead of GPU, impacting performance and energy consumption [156]. We discard it because the occurrences of this smell depend neither on the developer nor programming languages. The smell *Invalidate Without Rect* (IWR) [114] appears when the *onDraw* method is not implemented properly, resulting in overdraw views [156]. When developers do not specify the rectangle area that should be updated, the whole view is redrawn, even some area that is not visible, resulting in performance problems. Ni-Lewis [156] indicated that developers should call the method *invalidate(Rect dirty)*, specifying the area to be drawn, to avoid this smell. However, this method was deprecated in API 28 and since API 21 its calls are ignored. Consequently, we discarded this smell.

[2] `https://developer.android.com/about/dashboards/` Last visit: 06/11/2018

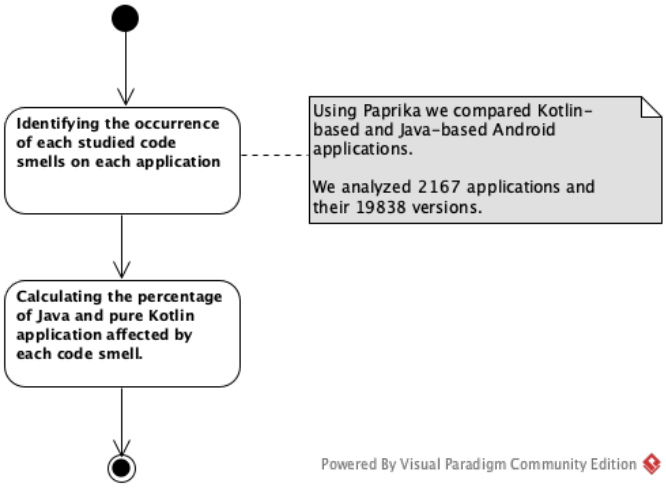


Figure 4.1: Steps followed to compare Java and pure Kotlin applications in terms of code smells.

4.2 Comparing the quality of Kotlin-based and Java-based Android applications

According to Ray et al. [230], language design affects software quality. Therefore, the choice between Java or Kotlin for developing Android applications may affect these applications' quality. For that reason, in this section, we compare the quality of Kotlin-based and Java-based Android applications in terms of code smells by answering our first research question:

- RQ_3**:** Is there a difference between the quality of Kotlin and Java Android applications, expressed in terms of code smells presence?

Figure 4.1 illustrates the steps followed to conduct this study. In Section 4.2.1, we describe the set of applications analyzed to answer this research question. Section 4.2.2 presents the methodology applied to obtain the results reported in Section 4.2.3.

4.2.1 Applications analyzed in the study

In this empirical study, we want to compare Kotlin-based and Java-based Android applications' quality in terms of code smells. For that reason, we used in this study all applications that we collected to investigate the adoption of Kotlin by Android developers (Section 3.1.1). Therefore, we analyzed 2 167 applications and their 19 838 versions (apks).

4.2.2 Analysis method

To identify the object-oriented and Android smells listed in Section 4.1.2.1, we analyzed the 2 167 applications. We ran Paprika for all versions of each application (i.e., apks). Then, we computed the percentage of Java and pure Kotlin (i.e., applications that are entirely written in Kotlin, previously identified in Section 3.2.3) applications affected by each code smell. We did not consider applications partially written in Kotlin in this study because it was not possible to distinguish whether a code at byte-code level comes from Java or Kotlin code.

An application a is affected by a code smells s if a has at least one instance of s. This approach splits applications into two groups: *a)* affected by s, and not affected by s. We also calculated a metric that computes the ratio between the number of instances of one code smell and the number of concerned entities related to this smell, as done by Habchi et al. [108]. The goal of this metric is to quantify the importance of the difference in proportions of the smells. Thus, for each application a, the ratio of a smell s is defined as [108]:

$$ratio_s(a) = \frac{fuzzy\_value_s(a)}{number\_of\_entities_s(a)} \tag{4.1}$$

where $fuzzy\_values_s(a)$ is the sum of the fuzzy values (which vary from 0 to 1) of the detected instances of a smell s in an app a. $number\_of\_entities_s(a)$ is the number of the entities concerned by the smell s in the app a. Section 4.1.2 shows the relationship between the smells and the entities concerned by them.

We computed Cliff's δ [236] as well, which indicates the magnitude of the effect size [54] of the treatment on the dependent variable. In our study, we used Cliff's δ to determine for each smell s which group of applications, pure Kotlin or Java applications, have more entities affected by s, and whether the effect size found presents a significant difference.

The Cliff's Delta estimator can be obtained with Equation 4.2.

$$\delta = \frac{\#(x_1 > x_2) - \#(x_1 < x_2)}{n_1 * n_2} \tag{4.2}$$

In this expression, x_1 and x_2 are scores within group 1 (Kotlin applications) and group 2 (Java applications), and n_1 and n_2 are the sizes of the sample groups. The cardinality symbol # indicates counting. This statistic estimates the probability that a value selected from one of the groups, in our case a $ratio_s$, is greater than a value selected from the other group, minus the reverse probability [166].

According to Romano et al. [236], the effect size is small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$. We opted for Cliff's δ test since it is suitable for non-normal distributions. Moreover, Cliff's δ is also recommended for comparing samples of different sizes [166].

4.2.3 Results

Android applications may vary in different aspects related to software size like the number of classes, the number of lines of code, the number of methods. For that reason, we initially analyze the proportion of applications affected and not affected by code smells. Then, in the second part of this study, we perform a more in-depth analysis by comparing the proportion of applications' entities affected by code smells. Using this second approach, we can normalize the occurrences of code smells according to Android applications' size and, consequently, we can compare Java and Kotlin applications fairly. The results presented in this section consider the 17 725 apks from 2 040 applications (94%) successfully analyzed using Paprika.

4.2.3.1 Number of Affected Applications

Table 4.2 shows for each programming language and code smell, the percentages of Android applications affected by a given code smell, i.e., having one or more smell instance.

We found that 3 out of 4 (75%) object-oriented smells affect more than 93% of applications considering both languages, Kotlin and Java. LM is the most common smell, affecting approximately 99% of the applications of both languages. SAK is the least frequent smell, but it

Table 4.2: Percentage of Android applications affected by code smell. An app *a* is affected by a code smell *s* if *a* has at least one instance of *s*.

	% Affected applications by smells									
	Object-oriented smells				Android smells					
	Long Method (LM)	Complex Class (CC)	BLOB class (BLOB)	Swiss Army Knife (SAK)	No Low Memory Resolver (NLMR)	UI Overdraw (UIO)	Heavy Broadcast Receiver (HBR)	Heavy Service Start (HSS)	Heavy ASynctask (HAS)	Init OnDraw (IOD)
Kotlin	99.80	98.50	95.12	65.67	99.30	54.02	35.72	17.31	09.55	09.55
Java	99.62	96.89	93.53	66.51	98.84	45.33	39.98	19.92	22.61	06.50
K - J	0.18	1.61	1.59	−0.84	0.46	8.69	−4.26	−2.61	−13.06	3.05

still affects most applications, around 65% of Kotlin applications and 66% of Java applications. Therefore, our results agree with previous work [115, 108] showing that LM, CC, BLOB and SAK are the most common OO smells in Android applications, respectively.

Table 4.2 also shows differences between the percentages of applications written in Java and Kotlin (row K - J): for all OO smells (LM, CC, BLOB, and SAK), the differences are small: 0.18%, 1.61%, 1.59% and -0.84%, respectively. Our conclusion is twofold. First, 3 out of 4 (75%) of object-oriented code smells are more present in Kotlin applications. However, they affect a similar proportion of both Kotlin and Java applications.

Regarding Android smells, we observed that NLMR is the most frequent smell, affecting the 99% of Kotlin and 98% Java applications. Furthermore, the second most present Android smell, UIO, affects 54% and 45% of Kotlin and Java applications, respectively. The third most present smell is HBR, which affects 35% of Kotlin and 39% of Java applications. Other 3 smells (HSS, HAS, IOD) are present in, at most, 22% of all applications. Note that 3 Android smells affect proportionally more Java applications. Moreover, the most significant difference between the proportion of applications affected by Android smells is 13.06% from the HAS smell.

We observed a high number of applications affected by NLMR smell because it is related to *Activities*, the main component of Android applications, present in almost every Android application. HBR, HSS and HAS are related to other Android components, *BroadcastReceiver*, Service and *AsyncTask*, that are not essential for all Android applications. UIO and IOD are smells related to *View* component, affect only custom views implementation. However, as the Android SDK provides several views implementations, most applications do not need custom views implementation.

Response to RQ_3**:** *Is there a difference between the quality of Kotlin and Java Android applications, expressed in terms of code smells presence?*
In terms of affected applications: Considering all smells, 6 out of 10 (3 out of 4 object-oriented and 3 out of 6 Android) smells affect proportionally more applications with Kotlin code. However, for all object-oriented smells and 4 out of 6 Android smells, the difference between the percentage of affected Kotlin and Java applications is small (between 0.18% and 4.26%).

This result shows that 3 out of 4 object-oriented smells and 3 out of 6 Android smells affect more Kotlin applications. However, we can not affirm that Kotlin applications are worse than

Java applications in terms of the presence of code smells because, in this analysis, we do not consider the number of code smells that each application has. Thus, one application with 100 instances of a given code smell is counted in the same way that an application has only one instance of the same code smell. For that reason, we also analyze the number of applications' entities affect by code smells.

4.2.3.2 Number of Affected Entities

Now, we study the proportion of entities affected by smells using the methodology presented in Section 4.2.2. In the previous section, we identify the proportion of Kotlin-based and Java-based applications affected by code smell. However, we ignore how much they are affected, i.e., we do not consider the density of code smells into applications. On the other hand, in the section, we analyze the density of code smells into applications by measuring the proportion of applications' entities affected by code smells' instances. Table 4.3 shows the median (med) ratios of smells in applications (Formula 4.1) and Cliff's δ effect sizes for each smell and programming language.

First, we observed that the most frequent smells (LM, CC, and BLOB) have a small median ratio. This means that, although they are present in most applications, only a few entities are affected by these smells. Moreover, Cliff's δ values show that the differences between the smell median ratio of pure Kotlin and Java applications are statistically significant for all object-oriented smells: 'small' difference for CC and SAK, and 'medium' for LM and BLOB. We conclude that although 3 out of 4 object-oriented smells affect more Kotlin applications, our results show that for 100% of object-oriented smells, Java applications have in median more entities affected by them with statistical relevance.

Concerning Android smells, we observed that the median ratios for 4 out of 6 smells (HBR, HSS, HAS and IOD) are zero for Java and Kotlin applications, which is consistent with Table 4.2 since less than 50% of applications for both languages are affected by these smells. Furthermore, Cliff's δ showed no significant difference for 5 out of 6 Android smells, including those previously mentioned. We conclude that very few entities are affected by these smells, even for HBR that affects more than 35% of Android applications (see Table 4.2). Concerning the smell NLMR, we observed, with statistical significance, that Java applications have more entities affected with a 'small' significant difference. This result agrees with those from Habchi et al. [108] and shows that NLMR affects most Java applications' activities.

Response to RQ_3**:** *Is there a difference between the quality of Kotlin and Java Android applications, expressed in terms of code smells presence?*
In terms of affected entities: Although 3 out of 4 (LM, CC, and BLOB) object-oriented smells affect more Kotlin applications, our results show that for all object-oriented smells (LM, CC, BLOB and SAK), Java applications have in median more entities affected by them with statistically relevance. Moreover, considering Android smells, Java applications have, in the median, more entities affected by NLMR, the unique Android smell that presents statistical relevance.

The results showed that in general, Java applications have more entities affected by all (4) object-oriented and one Android code smells. Therefore, by choosing Kotlin instead of Java, developers are minimizing the occurrences of these code smells in their applications.

Table 4.3: Ratio comparison between Kotlin and Java. The column 'Cliff's δ' shows the difference between the smell median ratio of pure Kotlin and Java applications: negative values mean that a smell affects fewer entities in Kotlin than in Java.

Smell	Lang	Median Ratio	Cliff's δ	Significance of difference
LM	Kotlin	0.0563	-0.3873	Medium
	Java	0.0736		
CC	Kotlin	0.0593	-0.3168	Small
	Java	0.0781		
BLOB	Kotlin	0.0163	-0.4338	Medium
	Java	0.0278		
SAK	Kotlin	0.0008	-0.2433	Small
	Java	0.0040		
NLMR	Kotlin	0.3750	-0.2915	Small
	Java	1.0000		
UIO	Kotlin	0.0769	0.1156	Insignificant
	Java	0.0000		
HBR	Kotlin	0.0000	-0.0699	Insignificant
	Java	0.0000		
HSS	Kotlin	0.0000	-0.0240	Insignificant
	Java	0.0000		
HAS	Kotlin	0.0000	-0.1306	Insignificant
	Java	0.0000		
IOD	Kotlin	0.0000	0.0341	Insignificant
	Java	0.0000		

4.3 Measuring the impact on the quality of introducing Kotlin

In Section 3.2.3, we found that 162 out of 244 (66.39%) applications with Kotlin code also have Java code. As Kotlin became an Android official programming language later than Java, potentially these applications were initially written in Java, and Kotlin has been added to it later. However, we do not know whether this introduction affects the applications' quality. In the section, we investigate the impact of adopting Kotlin on the quality of Android applications written in Java by answering our fourth research question:

- RQ_4**:** How frequent does the introduction of Kotlin positively impact on the quality of the versions of an Android application?

Figure 4.2 illustrates the steps executed during this study. Section 4.3.1 presents the target applications of this empirical study. Section 4.3.2 defines the quality model used to estimate the quality of Android applications. Section 4.3.3 explains how we trained this model. Section 4.3.4 describes how we used our quality model to answer RQ_4 and Section 4.3.5 presents the results of our empirical study.

4.3.1 Applications analyzed in the study

In order to investigate the impact of adopting Kotlin on the quality of Android apps, we need applications with two characteristics: i) they were initially written in Java and ii) later, they had Kotlin code introduced. For that reason, we target the 244 applications from FAMAZOA, a dataset of open source Android applications written in Kotlin (Section 3.1.3). These applications have Kotlin code, but they were not necessarily written from scratch using Java. For instance, some of them may have only Kotlin code. Applying the methodology presented in Section 4.3.4, we consider only the applications that have both characteristics mentioned previously.

4.3.2 Defining a quality model

For responding to RQ_4, we used the technique presented by Hecht et al. [115] for scoring each version (apk) from a mobile application. The score serves as an estimation of the mobile app quality in a particular version (apk) and is based on the consistency between the application's size and the number of detected code smells.

To compute the software quality score based on one type of code smell s, the technique from Hecht et al. [117] first builds an estimation model using linear regression. This model represents the relationship between the number of code smells of type s and one metric representing the size of the application in terms of the numbers of entities associated with s. For example, for the smell BLOB, the metric of size that we consider is the total number of classes, and for the smell LM, the metric is the number of methods. Table 4.1 presents the relation between the types of smells and entities. We built a quality model for each code smell that we consider in this study (Section 4.1.2.1).

To obtain the quality score for one application (apk), the technique takes as input the number of code smells and a value of that app's size, and produces as output a score. A higher positive score implies better quality. As described by Hecht et al. [115], the quality score of an application at a particular version is computed as the additive inverse of the residual. Consequently, a larger positive residual value suggests worst software quality because it means the apk has more smells with respect to its size than the norm (i.e., linear regression). In contrast, a larger negative residual value implies better quality because of the lower number of smells.

4.3.3 Training a quality model

We created a quality score model, i.e., a linear regression, for each code smell presented in Section 4.1.2.1. We trained the linear model using a dataset defined by Hecht et al. [115] which contains 3 568 Android versions (apk) extracted from the Google Play store between June 2013 and June 2014. We selected this dataset for training the model for two main reasons: *1)* it was previously used in a similar studies for training quality models Hecht et al. [115] and Hecht [114]; and *2)* its applications do not include Kotlin code. The trained model created from this dataset represents the quality of applications built previously to Kotlin was released. Thus, we used it as a baseline to measure if Kotlin applications have more (or less) quality than applications written before Kotlin was released.

To create a quality model, we first run Paprika over the dataset previously mentioned. The output of Paprika is the training set. Each element of the training dataset (a row) corresponds to a single apk *a* and has the following information: *a)* the number of instances of a smell *s* in application *a*, and *b)* the value associated with the entity of smell *s* in *a*. For example, for smell Long Method (LM) we compute the linear regression between: *1)* the number of instances of smell LM,[3] and *2)* the total number of methods.

4.3.4 Analysis method

Once we trained our quality model, we computed the quality scores (one per code smell) for each apk from our dataset of applications classified as 'Kotlin', i.e., applications that mix Java and Kotlin previously identified in Section 3.1.3. Those apks conform to our *test* dataset. Note that the training dataset does not include any apk from the *test* dataset. Thus, we discarded the possibility of having overfitting in our models.

In this study, we analyzed the applications that initially have one or more apks classified as Java and then it has 1 or more apks classified as Kotlin. We measured the impact on the quality of introducing Kotlin code in one application as follows. For each of those applications and for each code smell *s*, we first compared the quality scores of *s* between the apk that introduces Kotlin code and the previous apk (i.e., which has Java code and no Kotlin). Then, we compared the quality score between the last Java apk (i.e., the version just before introducing of Kotlin code) and the most recent Kotlin version available. These two comparisons have different goals: the first one aims at measuring the impact just after the introduction of Kotlin in an app; the second one aims at studying the impact after the application (that now includes Kotlin code) has evolved.

Using this information, we studied the quality evolution trend of each application. A quality evolution trend describes how the quality scores from the versions of an application *a* change throughout its evolution. Considering 5 major quality evolution trends defined by Hecht et al. [115]: *A)* Constant decline, *B)* Constant rise, *C)* Stability, *D)* Sudden decline, and *E)* Sudden rise, we verified whether the introduction of Kotlin code into an app *a* produces a *positive* change in the quality evolution trend.

For each application *a*, we classified the quality evolution trend manually *before* and *after* the introduction of Kotlin on *a*. Then, we considered that the introduction of Kotlin produces a *positive* change if: *1)* the trend before the introduction is 'Decline' or 'Stability' (trends A, C or D); and *2)* the trend after the introduction is exclusively 'Rise' (trends B or E). Note that we discarded analyzing those applications whose trends (before or after) do not fit any defined trends.

[3] Hecht [114] considers that a method is "long" (LM) if it has more than 17 instructions.

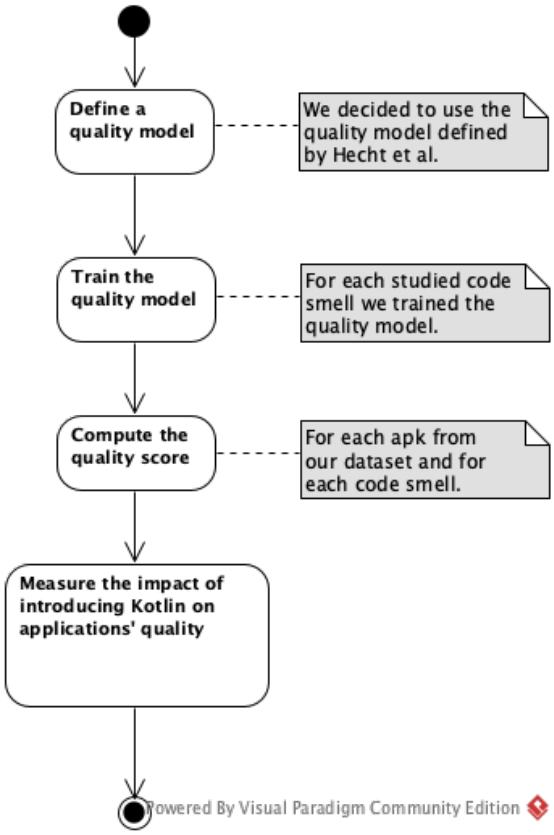


Figure 4.2: Steps followed to measure the impact of introducing Kotlin on applications' quality.

Table 4.4: Changes on quality scores after introducing Kotlin.

Code Smell	# Apps Kotlin Improves Quality Score		Positive Change
	First Kotlin	Last Kotlin	on Evolution Trend
LM	28 (50%)	28 (50%)	10/28 (35.71%)
CC	35 (62.5%)	36 (64.29%)	13/36 (36.11%)
BLOB	43 (76.79%)	42 (75%)	18/42 (42.86%)
SAK	44 (78.57%)	45 (80.36%)	17/45 (37.78%)
HBR	45 (80.36%)	40 (71.43%)	7/40 (17.5%)
HAS	37 (66.07%)	30 (53.57%)	2/30 (6.67%)
HSS	45 (80.36%)	42 (75%)	6/42 (14.29%)
IOD	36 (64.29%)	36 (64.29%)	5/36 (13.89%)
NLMR	33 (58.93%)	31 (55.36%)	6/31 (19.35%)
UIO	36 (64.29%)	36 (64.29%)	8/36 (22.22%)

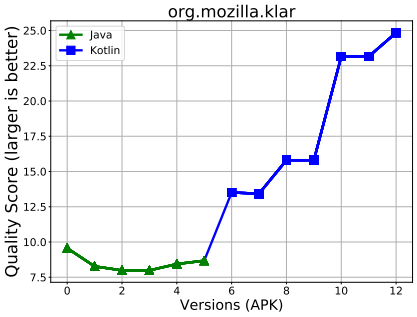
4.3.5 Results

In our empirical study, we found a total of 57 applications that initially had only Java code and later introduced Kotlin. Considering these applications, Table 4.4 shows the number of Kotlin applications whose quality score increases after the introduction of Kotlin code for each type of smell. The increase of one quality score associated with one type of smell implies fewer instances of that smell and, consequently, a better quality of the application [115].

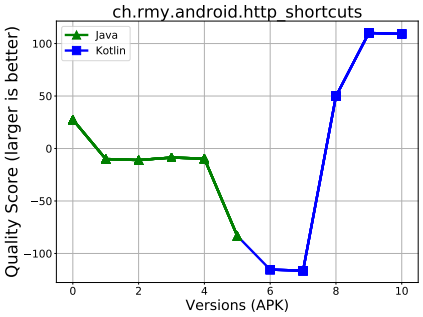
The results show that for the 10 smells, for at least the 50% of the applications that introduced Kotlin code, their quality scores increased between the last Java version and the first version with Kotlin (see Table 5.1 column 'First Kotlin'). That means, for such applications, the introduction of Kotlin code impacted the quality scores positively. Note that 50% of values is the lower bound value because smell LM improves precisely the 50% of applications. However, all the other smells improve more than 50%. For instance, Table 5.1 shows that, for 8 out of 10 smells, the percentage of applications with quality improvement is larger than the 62% and for 4 smells is larger than 76%. That means that for only one smell (LM), 50% of apps do not improve the quality. For all other smells, the number of apps with improvement is larger than the number of apps that do not enhance. Furthermore, for all smells, at least the 50% applications improved the quality score between the last Java version and the most recent (i.e., the last) Kotlin version (see Table 5.1 column 'Last Kotlin').

For instance, let us focus on smell CC (complex-class) at the second row of Table 4.4. As column "First Kotlin" shows, for 35 out of 56 (62.5%) applications, the version v_k that introduces Kotlin code has greater (i.e., better) quality score associated with the CC smell than the last version without Kotlin v_{k-1} (i.e., the previous version of v_k). Figure 4.3a shows the quality score associated with the CC smell of each version of "Mozilla Klar" app. The last version that does not contain Kotlin code corresponds to X=5 in that figure. We observed that the first version that has Kotlin code (X=6) increases the quality score, as well as all the subsequent versions (X=[6..12]) do.

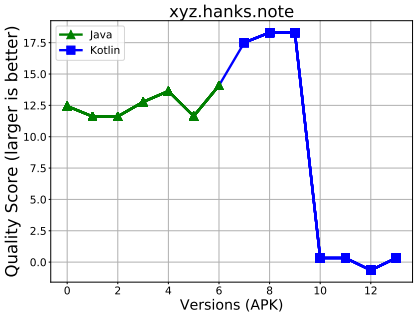
Furthermore, as the column "Last Kotlin" shows, for 36 applications (64.29%), the most recent version with Kotlin code has a greater (i.e., better) quality score associated with the CC smell than the version before the introduction of Kotlin. Again, "Mozilla Klar" is one of those applications: the last version (X=12) has a higher score than the last Java version (X=5). Note that, for the CC smell, there is 1 application (36 - 35) whose quality scores: *a)* decreases in the version that introduces Kotlin, but *b)* increases in the most recent Kotlin version. The evolution



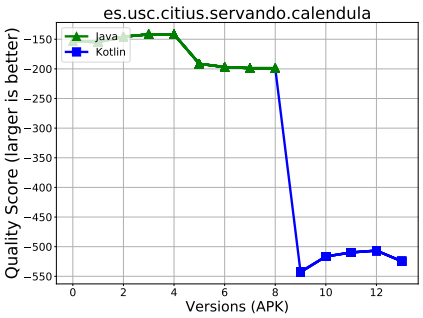
(a) Improvement of the first and last Kotlin versions.



(b) Improvement of the last Kotlin version.

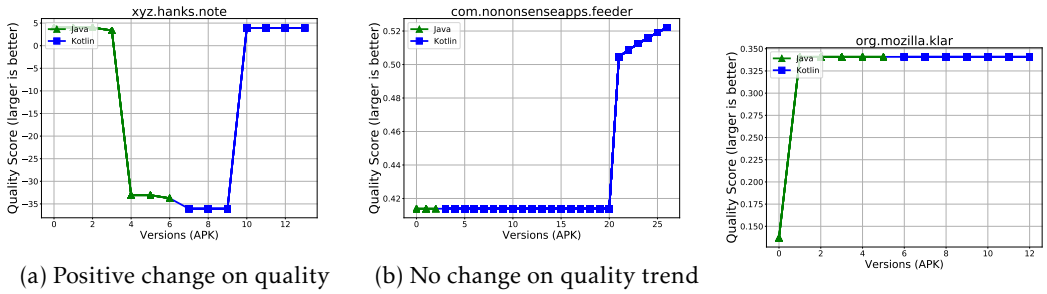


(c) Improvement of the first Kotlin version.



(d) No improvement after introducing Kotlin.

Figure 4.3: Evolution of quality scores based on CC smell along the version history.



(a) Positive change on quality trend (smell NLMR). Trend before: 'Decline', trend after: 'Rise'.

(b) No change on quality trend (smell HAS). Trend before: 'Stability', trend after: 'Stability' (even later is 'Rise').

(c) No change on quality trend (smell HBR). Trend before: 'Rise', trend after: 'Rise'.

Figure 4.4: Three cases of classification of quality trend evolution *before* and *after* the introduction of Kotlin code.

of the quality score of one of those applications, named "HTTP Shortcuts", is displayed in sub-figure 4.3b.

We also observed in Table 4.4 that for 5 out of 10 smells, the number of applications with quality improvements after the first Kotlin version (column "First Kotlin') is larger than the number of applications with quality improvements over the last Kotlin version (column "Last Kotlin'). This means that some Kotlin applications' quality scores decrease between the first and the last Kotlin versions. For instance, sub-figure 4.3c shows the quality score evolution of "Hanks Note" application: the first Kotlin version increases the score w.r.t of the last Java version. However, during the subsequent versions, the quality scores drop, even lower than the last Java version's score. Finally, sub-figure 4.3d shows the quality score evolution of one application, named "Calendula", whose quality score constantly decreases after introducing Kotlin code.

This finding shows that also the quality of Kotlin applications can be degraded throughout the app evolution.

Response to RQ_4**:** *How frequent does the introduction of Kotlin positively impact on the quality of the versions of an Android application?* The introduction of Kotlin code in Android applications initially written in Java produces a rise in the quality scores from, at least, the 50% of the Android applications. More precisely, for 8 out of 10, the first commits with Kotlin code produce a rise in the quality in at least 62% of the studied applications.

Finally, the last column from Table 4.4 shows the number of applications where the introduction of Kotlin has changed the quality evolution trend from 'Decline' or 'Stability' to 'Rise' (Section 4.3.4). We called those *Positive changes* on quality evolution changes. Regarding the object-oriented smells, the percentage of applications that improved their quality between the last version of Java and the last of Kotlin varies between the 35.7% and 42.8%. For the Android smells, the number of applications with positive change is lower: between 6.6% and 22.2%.

Figure 4.4 shows three cases. The first one, displayed in sub-figure 4.4a, corresponds to a positive change in the quality evolution trend. Before the introduction of Kotlin, the quality scores were constantly declining. The introduction of Kotlin has positively changed the evolution trend: after that, the quality scores constantly rise. The second case, sub-figure 4.4b, shows that the introduction of Kotlin does not change the trend: the quality score before and after the introduction is stable. Note that, over the end of the evolution, the quality score suddenly

rises. However, we did not associate this rise with the introduction of Kotlin, which was done much before. Finally, the third case, sub-figure 4.4c, does not present a change on the trend: the quality score was rising before the introduction of Kotlin and continues rising after that.

In conclusion, the study about the changes of quality evolution trend showed that some applications: *a)* stopped having a constant degradation of the quality of the app written in Java, and *b)* presented an improvement of quality that rises after the introduction of Kotlin code.

4.4 Threats of validity

4.4.1 Internal

Modelization of software quality models. There is a risk that the training dataset could not be representative. Thus, the quality model produces incorrect estimation. We consider the same training dataset that previous work has used to create quality models and detect smells on Android applications [115, 117, 116]. Furthermore, the use of that dataset allows having a training dataset and a validation dataset without any intersection, avoiding the generation of an overfitted model.

The set of smells studied. To compare fairly the presence of code smells in Java and Kotlin applications, we selected Paprika tool, which works at the bytecode level. However, other code smells that are not considered in our work could impact the code's quality. Nonetheless, according to Mannan et al. [174], 3 out 4 object-oriented code smells considered in this work are in the ranking of the most studied code smells in Android applications.

Furthermore, Kotlin provides new features and different syntax that could introduce new types of code smells. Nevertheless, to the best of our knowledge, there are no studies in the literature that investigate the impact of the adoption of Kotlin, and consequently its features, on the source code quality.

4.4.2 External

The validity of Paprika. It could exist the risk that Paprika has *a)* false positives, i.e., it detects smells instances that are not correct, and *b)* false negatives, i.e., it does not detect smell instances. However, Paprika has been extensively evaluated through different experiments [114, 115, 117, 116].

Applications' Representativeness. Our empirical study focus on studying the bytecode of Java-based and Kotlin-based Android applications. For that reason, we used the applications collected in our previous study (Section 3.1.1), because to the best of our knowledge, it is the largest set of open source Android applications that contains apks from applications written in Java and Kotlin. However, we cannot generalize our findings over applications that are not open source.

Missing versions (apks). In our previous study, we collected apk of the application hosted on F-droid and AndroZoo. However, there is a risk that F-droid and AndroZoo do not contain all the released versions of an Android application. Consequently, this missing data could affect our analysis of the application quality, which is based on the analysis of all apks available of F-droid.

Kotlin learning curve In this study, we did not consider the developers' experience with Kotlin and consequently, the effect of the learning curve of Kotlin in the source code written using this programming language. As Kotlin is a new programming language compared to Java, especially in the context of Android development, it is possible that some code smells found results of the lack of experience in Kotlin. Therefore, by targeting experienced Kotlin developers, we could find a different result. Nevertheless, all applications analyzed were published on F-droid or Google Play, then they represented snapshot of Android development in the period of our study.

4.5 Summary

In this chapter, we presented an empirical study investigating how the adoption of Kotlin impacts the quality of Android applications. Initially, we compared the occurrence of ten different code smells, 4 object-oriented and 6 Android-specific, on Java-based and Kotlin-based Android applications. Then, using a quality model that estimates the quality of applications' version based on the presence of code smells, we analyzed how the introduction of Kotlin affects the quality evolution trend of Android applications initially written in Java.

In this empirical study, we used Paprika to identify the presence of code smells on Android applications. Paprika identifies the number of instances of a given code smell and the number of affected entities related to each smell. Using this information, we compared two sets of Android applications: applications written in Java and applications written in Kotlin. Moreover, using the information about the occurrence of code smells, we estimated its quality score for each application version using a quality model based on linear regression defined by Hecht et al. [115]. Then, considering applications initially written in Java, we evaluate the impact of the adoption of Kotlin using this quality model.

We found that 3 out of 4 object-oriented smells (LM, CC and BLOB) are present in, at least, the 93% of both Java and Kotlin applications. In percentage, 3 out of 4 object-oriented smells (LM, CC and BLOB) are more frequent in Kotlin applications. However, we found that Java applications have more entities affected by 5 out of 10 code smells, including all object-oriented smells. Moreover, we found that the introduction of Kotlin code on an Android application written in Java produced a rise of the quality on, at least, the 50% of the studied applications. Therefore, these results showed that, in general, the adoption of Kotlin impacts the quality of Android applications positively.

This work allowed us to show that, in general, choosing Kotlin instead of Java impacts the quality of Android applications positively. Therefore, this is a motivation for developers to add Kotlin to their applications or even migrate them. However, at this point, it is not clear how developers are introducing Kotlin. For instance, whether Android developers are migrating applications written in Java or adding new features using Kotlin or how this code evolves. Moreover, we also do not know if developers have been using the programming language features provided by Kotlin. In Chapter 5, we investigated the evolution of Kotlin code in Android applications and the usage of Kotlin features by Android developers.

Chapter 5

Analyzing the evolution of Kotlin code in Android applications

In Chapter 4, we showed that adding Kotlin code to Android applications initially written in Java improved the quality, in terms of code smell presence, for most of the studied applications. As Kotlin is fully interoperable with Java, it implies that developers can introduce Kotlin code into their Java-based applications without migrating that existing code. Thus, an application written initially in Java can evolve, for instance, by adding Kotlin gradually. Moreover, Kotlin provides a different approach to write applications because it combines object-oriented and functional features, some of them not present in Java or not available for Android development [57]. Even though, as presented in Section 3.1.3, around 11% of Android applications have adopted Kotlin. Although some studies were published about Kotlin, to the best of our knowledge, there was no study in the literature about the evolution of Kotlin code in Android.

For that reason, the goal of this chapter is to understand the evolution of Kotlin code in Android applications. To that end, we conduct two empirical studies. First, we investigate how the amount of Kotlin code changes over the evolution of Android applications. For instance, we want to know whether developers continue adding Java code after introducing Kotlin. Then, in the second study, we focus on the language features introduced by Kotlin, such as *Smart cast*(See Appendix B.8). In particular, we explore four aspects of Kotlin features usage: *i)* **which** features are adopted, *ii)* **what** is the degree of adoption, *iii)* **when** are these features added into Android applications for the first time, and *iv)* **how** the usage of features evolves along with applications' evolution. These empirical studies answer the following research questions:

- RQ_5**:** How does code evolve along the history of an Android application after introducing Kotlin code?
- RQ_6**:** Which Kotlin features are adopted by Android developers?
- RQ_7**:** When do Android developers introduce Kotlin features during applications' evolution?
- RQ_8**:** How the usage of Kotlin features evolves along with the evolution of Android applications?

To carry out these empirical studies, we used FAMAZOA, the largest publicly available dataset of open source Android applications written in Kotlin (Section 3.1.3). In the first study,

we analyzed each application's commits to measure the amount of Java and Kotlin code. We use this information to define the observed code evolution trends. Using these trends, we classify each application according to the trend that better describes Kotlin code's evolution. For example, one trend is that Java is removed totally, whereas Kotlin is added. The second study is a finer-grained study of code, which focuses on the usage and evolution of Kotlin features. To investigate these features' adoption, we extracted them from each version of applications' source code, and we identified when features were used for the first time. To understand how the use of features evolves, we analyzed each application's code repository (i.e., Git) to mine the features used on each version (commit). Then, we inspect the most frequent usage trend for each Kotlin feature.

The chapter continues as follows. Section 5.1 presents our empirical study about the evolution of Kotlin code in Android applications. Section 5.2 presents our empirical study about the usage and evolution of Kotlin features in Android applications. Finally, Section 5.3 summarizes our work and outlines further works to perform on this research topic.

This chapter includes a revised version of two papers published in the proceedings of ESEM 2020 [184] and the Empirical Software Engineering journal [99].

5.1 Analyzing the code evolution of Android applications

Chapter 3 showed that Android developers are using Kotlin to build Android applications. However, it is not clear whether developers are using Kotlin to create new applications from scratch or add new functionalities using Kotlin, or even if they are migrating Java code to Kotlin.

In this section, we investigated the evolution of Kotlin code in Android applications by answering the first research question:

- RQ_5**:** How does code evolve along the history of an Android application after introducing Kotlin code?

To answer this research question, we conducted an empirical study that analyzes the source code of Android applications. Figure 5.1 illustrates the steps followed to conduct this study. Section 5.1.1 describes the set of Android applications analyzed in this empirical study. Section 5.1.2 introduces the evolution trends of code considered in this empirical study. Section 5.1.3 explains the methodology applied to investigate the evolution of Kotlin code in Android applications. Section 5.1.4 presents the result of this empirical study. Finally, Section 5.1.5 outlines threats of validity.

5.1.1 Applications analyzed in the study

This empirical study aims to investigate how the amount of Kotlin evolves throughout applications' development. Consequently, to respond to this research question, we need Android applications that contain Kotlin code. For that reason, we target applications from FAMAZOA, a dataset of open source Android applications written in Kotlin, presented in Section 3.1.3. FAMAZOA has 244 Android applications totally of partially written in Kotlin.

5.1.2 Code evolution trends

We defined 12 cases that represent different evolution trends of Kotlin and Java code. The procedure we used to define those trends is as follows: First, we plotted a two-dimension plot for each application where the axis X corresponded to the number of commits (chronologically

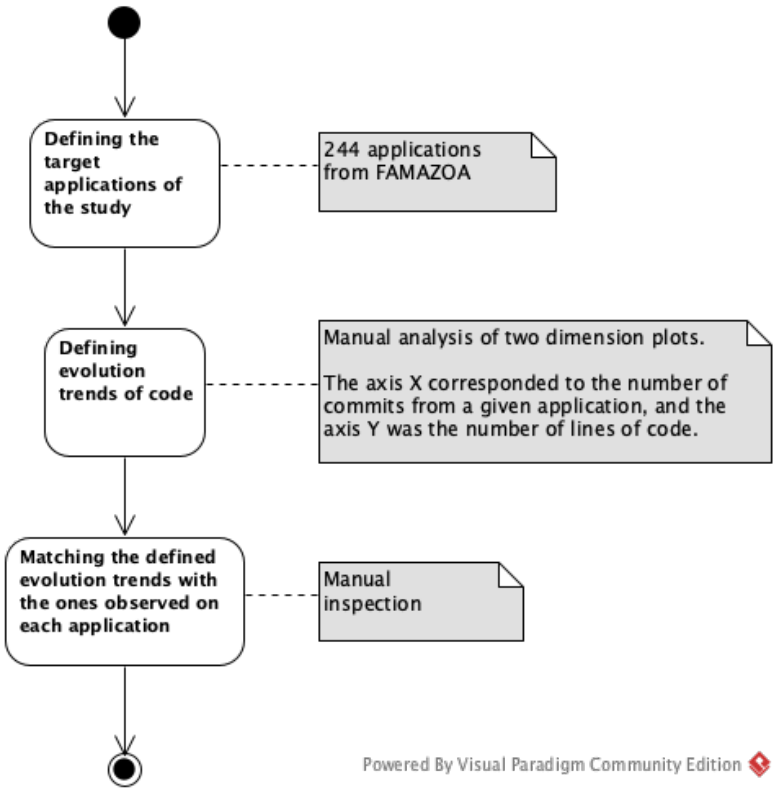


Figure 5.1: Steps executed to perform the analysis of the code evolution trends of Android applications.

ordered) from a given application, and the axis Y was the number of lines of code. Then, we plotted two series, one corresponding to the number of code lines written in Java and one corresponding to the number of code lines written in Kotlin. For each plot, we described the trend we observed. Then, we checked whether a similar trend had been seen before based on such descriptions. To avoid bias, both authors (myself, a Ph.D. student and my Ph.D. co-advisor) have analyzed the same plots separately, and later, we compared and discussed the result found.

The trends identified are:

ET 1: Kotlin is the initial language, and the amount of Kotlin grows throughout history.

ET 2: Kotlin code replaces all Java code between two consecutive versions.

ET 3: Kotlin code replaces some Java in consecutive versions (i.e., amount of Java code drops), but the amount of Java continues growing after the drop.

ET 4: Kotlin increases together with Java.

ET 5: Kotlin grows, and Java decreases, and the last version of the app has both languages.

ET 6: Kotlin grows, and Java decreases until the amount of Java code is 0.

ET 7: Kotlin grows, and Java remains constant.

ET 8: Kotlin is constant and Java changes.

ET 9: Kotlin and Java remain constant.

ET 10: Kotlin introduced in the app but lately disappears (the amount is 0).

ET 11: Java replaces Kotlin code.

ET 12: Other.

Note that it could exist more evolution trends that we have not included in the previous list. Those would be included in 'Other'. We included only those we have observed during our empirical study and are particularly interesting for this thesis.

5.1.3 Analysis method

We explore each application's source code repository to analyze Kotlin code's trend throughout the application history. Given a repository, we analyzed each commit in chronological order (i.e., starting from the oldest one) to calculate the amount of code (using CLOC) on that commit. We analyzed the evolution trend of two particular languages: Java (i.e., the traditional used for developing Android applications) and Kotlin.

To classify each Kotlin application according to its evolution trend, we first plotted the amount of code (lines of code) of Kotlin and Java for each commit. Figure 5.2 shows some of such plots. Then, we manually selected the most representative evolution trend (i.e., that better fits) to that application's code evolution. We classified an application with a given evolution trend only when both authors (myself, a Ph.D. student, and my Ph.D. co-advisor) of the paper fully agree on the classification. Otherwise, we classified the applications without consensus as *Other*. Moreover, we make publicly available in our appendix all the plots and the resulting classifications to further analyze our studies.[1]

[1] https://github.com/UPHF/kotlinandroid/tree/master/docs/evolution/

5.1.4 Results

In this section, we present the results obtained when applying the methodology presented in Section 5.1.3. Table 5.1 shows the results and Figure 5.2 displays, for each code evolution trend, the code evolution of one particular application as an example.

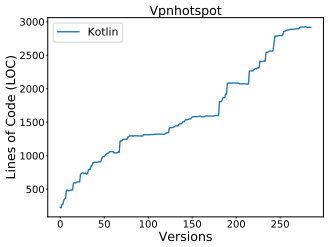
Table 5.1: Classification of Android applications according to the evolution trend of Kotlin and Java source code.

	Source Code Evolution Trend	# Apps	%
ET 1	Kotlin is the initial language and the amount of Kotlin grows	19	7.8
ET 2	Kotlin code replaces all Java code	15	6.1
ET 3	Kotlin code replaces some Java then Java continues growing	4	1.6
ET 4	Kotlin increase together with Java	8	3.3
ET 5	Kotlin grows and Java decreases, but it does not reach zero	52	21.3
ET 6	Kotlin grows and Java decreases until the Java code is 0	48	19.7
ET 7	Kotlin grows and Java remains constant	41	16.8
ET 8	Kotlin is constant and Java changes	43	17.6
ET 9	Kotlin and Java remain constant	7	2.9
ET 10	Kotlin introduced but lately disappears	3	1.2
ET 11	Java replaces Kotlin code	2	0.8
ET 12	Other	2	0.8
Total applications		244	100%

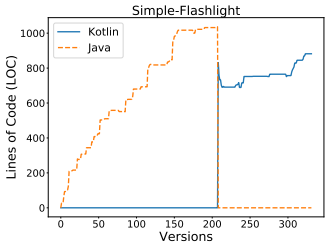
The most frequent code evolution trend we found is ET 5 (Kotlin grows and Java decreases, but it does not reach zero), with 52 out of 244 (21.3%) Kotlin applications. This evolution trend represents the cases that, after the first version (i.e., commit) that introduces Kotlin code, Kotlin code's amount tends to grow, whereas the amount of Java code decreases. Sub-Figure 5.2e shows the code evolution of the application *Jalkametri-Android*, which corresponds to that evolution trend. Still, the last version of *Jalkametri-Android* has more lines of code (LOC) of Java than Kotlin. Another application classified as ET 5 is *Poet-Assistant* (Sub-Figure 5.2f). However, unlike *Jalkametri-Android*, the amount of Kotlin code in the last version is larger than the amount of Java code. In the mentioned applications, the lines representing the evolution of Kotlin code seem to be symmetric with respect to those of Java. We suppose that in those cases, some Java components of these applications were gradually migrated to Kotlin code.

A similar trend to ET 5 is ET 6 (Kotlin grows and Java decreases until the Java code is 0): 48 Android applications (19.7%) exhibited that trend. Unlike ET 5, the amount of Java code in trend ET 6 gradually decreases until arriving at zero LOC. The Sub-Figure 5.2g shows one application, *Simple-Calendar*, whose first versions were written in Java. Then, in the versions from commit *09ef99* to *206dfe*, the authors added Kotlin and removed Java code. Finally, from commit *eee184* until its last commit, the application is only composed of Kotlin code.

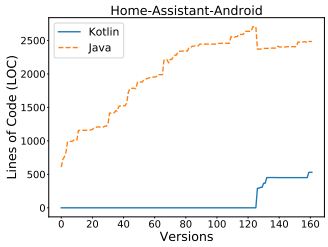
There are two trends, ET 7 (Kotlin grows and Java remains constant) and ET 8 (Kotlin is constant and Java grows) with 41 and 43 applications, respectively, which amount of code of one language remains stable after the introduction of Kotlin. For example, the Sub-Figure 5.2i shows the code evolution of *Talk-Android*, classified as ET 8. One commit (*7f12*) introduced a portion of Kotlin code (105 lines). Since then: *a)* the amount of Kotlin code remains constant throughout the evolution (in the last commit (#724) it has 106 LOC), *b)* the amount of Java code constantly grows. The Sub-Figure 5.2h shows an inverted case (app *Bimba*): the amount of Java code is constant while the amount of Kotlin code grows. There are also 7 applications whose amount



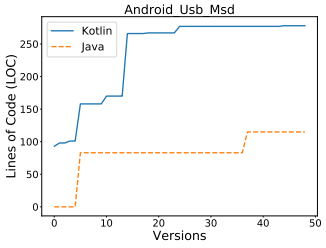
(a) ET 1: Kotlin is the initial language.



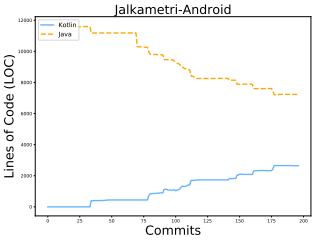
(b) ET 2: Kotlin code replaces all Java code.



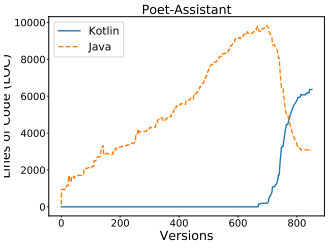
(c) ET 3: Kotlin code replaces some Java then Java continues growing.



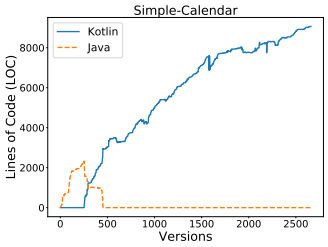
(d) ET 4: Kotlin increase together with Java.



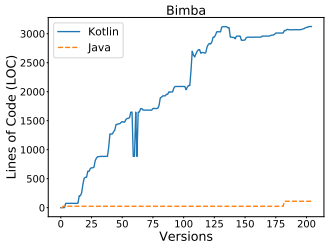
(e) ET 5: Kotlin grows and Java decreases. Example 1.



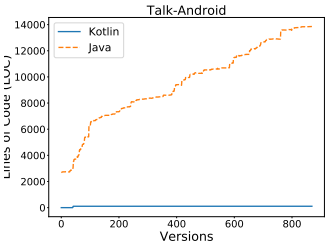
(f) ET 5: Kotlin grows and Java decreases. Example 2.



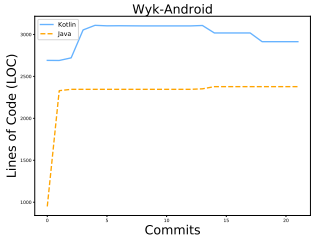
(g) ET 6: Kotlin grows and Java decrease until the Java code is 0.



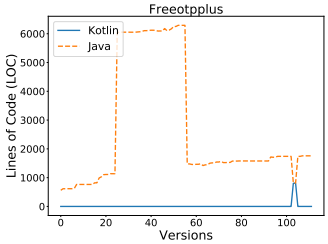
(h) ET 7: Kotlin grows and Java remains constant.



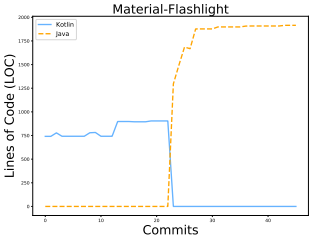
(i) ET 8: Kotlin is constant and Java grows.



(j) ET 9: Kotlin and Java remain constant.



(k) ET 10: Kotlin introduced but lately disappears.



(l) ET 11: Java replaces Kotlin code.

Figure 5.2: Evolution trends of Kotlin and Java code. Each plot presents the evolution of Kotlin and Java code along the history of one single application where the x-axis corresponds to the commits and the y-axis corresponds to the amount of code (i.e., lines).

of code written in both languages remain constant. Such as the app *Wyk-Android* showed in Sub-Figure 5.2j. These applications represent the trend ET 9 (Kotlin and Java remain constant).

The fifth most frequent evolution trend is ET 1 (Kotlin is the initial language), with 19 applications. Those applications were initially written in Kotlin and did not include Java code in any version. The application *Vpnhotspot* is one of them (Sub-Figure 5.2a). The evolution trend ET 2 (Kotlin code replaces all Java code), with 15 applications over 244 (6.1%), represents applications such *Simple-Flashing*, Sub-Figure 5.2b, which are initially written in Java and completely migrated in one commit (*18b5c9*). However, unlike ET 5 and 6, in ET 2 no versions share Java and Kotlin code. On the other hand, we classified 2 applications with trend ET 11 (Java replaces Kotlin code), i.e., applications whose evolution presents the opposite behavior compared with ET 2, such application *Material-Flashlight*, Sub-Figure 5.2l. These applications migrated from Kotlin to Java code.

Moreover, there were 8 Android applications (3.3%) that correspond to trend ET 4 (Kotlin increases together with Java): the amount of both Java and Kotlin grows. The Sub-Figure 5.2d shows the amount of code of the application *Android USB MSD*. The introduction of code written in one language did not produce a decrease in the amount of code written in the other language.

Another evolution trend is ET 3 (Kotlin code replaces some Java, then Java continues growing). When Kotlin code is introduced, the amount of Java code decreases (in similar proportions) but then the amount of Java starts growing again. Sub-Figure 5.2c shows the code evolution of application *Home-Assistant*, one of the 4 applications (1.6%) classified as ET 3. Here, we suspect that developers migrated only a portion of the code.

Furthermore, there were 3 applications represented by ET 10 (Kotlin introduced but lately disappears): Kotlin code is introduced at some time but, at some version later, that code is removed. Sub-Figure 5.2k shows the code evolution of the *Freeotpplus* application. We can see that Kotlin is introduced in the commit *2dbc32*, but later, after two commits, it is completely removed.

Finally, we assigned the trend ET 12 (Other) to 2 applications that we could not assign to any of our previously defined evolution trends. In our appendix, we presented the classification of the evolution trend discussed in this section.[2]

Response to RQ 5: *How does code evolve along the history of an Android application after introducing Kotlin code?*
For the 63.9% of the Kotlin applications, the amount of Kotlin code increases throughout the Android application evolution, and, at the same time, the amount of Java code decreases or remains constant (cases ET 2, 5, 6, and 7). For the 25.8% of applications, the Kotlin code replaces the totality of the Java code written on those applications (cases ET 2 and 6).

This result shows that once Kotlin is introduced into an application, it tends to become the dominant programming language, i.e., the language used to write most of its source code. Due to this trend of dominance, we concluded that it is necessary to understand how Android developers are using Kotlin features. Consequently, we decided to analyze the Kotlin code in a fine-grained manner: at the lever of code features. In Section 5.2, we present three empirical studies about the usage and evolution of Kotlin features in Android applications. Moreover, it shows a trend of migration of Android applications from Java to Kotlin since 25.8% (ET 2 + ET 6) of applications have migrated. Chapter 6 presents a machine learning model to assist developers that want to follow these trends, i.e., migrate their applications.

[2]Classification of the evolution trends. `https://github.com/UPHF/kotlinandroid/tree/master/docs/evolution`

5.1.5 Threats to Validity

5.1.5.1 External

Representativeness of FAMAZOA. Our work relies on FAMAZOA, a dataset of open source mobile applications written in Kotlin. Considering the number of applications published on Google Play, FAMAZOA represents a small parcel since it only contains open source applications, limiting the generalization of our findings. However, to the best of our knowledge, it is the largest dataset of Android open source applications written in Kotlin.

Comparison between Kotlin and Java code. To compare the amount of Kotlin and Java code on each application, we used the tool CLOC.[3] Consequently, our results are dependent on the preciseness of that tool. To avoid that, we manually checked a sample of results and did not find any issue.

5.2 The usage and evolution of Kotlin features

When Kotlin became an official programming language in 2017 [53], Android developers gained another option for creating their applications. Kotlin is a programming language that combines object-oriented and functional features, which brings to the development of Android applications programming language features not provided by Java such as *Coroutines*. Although some studies were published about Kotlin, to the best of our knowledge, there was no study in the literature focused on the usage of Kotlin features by Android developers.

As pointed by Mazinanian et al. [186], this lack of knowledge negatively affects four audiences: *i)* researchers are not aware of the research gaps (i.e., the actual unsolved problems faced by the developers) and thus miss opportunities to improve the current state of the art, *ii)* language and library designers do not know if the developers effectively use the programming constructs and APIs they provide are effectively used by the developers or are rather misused or underused, *iii)* tool builders do not know how to tailor their tools, such as recommendation systems and code assistants, to the developers' actual needs and practices when using Kotlin, *iv)* developers are not aware of the good and bad practices related to the use of Kotlin features.

For that reason, this section presents an empirical study that investigates different aspects related to the usage of Kotlin features by Android developers. We answer the following research questions:

- RQ_6: Which Kotlin features are adopted by Android developers?
- RQ_7: When do Android developers introduce Kotlin features during applications' evolution?
- RQ_8: How the usage of Kotlin features evolves along with the evolution of Android applications?

Figure 5.3 illustrates the steps executed to answer RQ_6, RQ_7 and RQ_8. Section 5.2.1 presents the study design applied to respond to these research questions. It presents the Kotlin features considered in our empirical study, the tool that we created to identify these features, and the dataset of Android applications used to investigate the usage of Kotlin features. Section 5.2.2 describes the study we conducted to investigate the adoption of Kotlin features. Section 5.2.3 describes the study that we conducted to understand when Android developers add Kotlin

[3] https://github.com/AlDanial/cloc

features into Android applications. Section 5.2.4 describes the study that aims to identify evolution trends of Kotlin features' usage. Section 5.2.5 presents a discussion about the result of this empirical study. Finally, Section 5.2.6 reports threats of validity.

5.2.1 Study Design

When a new programming language is released, it offers developers a set of language *features*. Other languages could already provide some of these features, whereas other features can be completely new. For example, the feature *Operator Overloading* is included in C++ (but not in Java). In this thesis, we exclusively focus on features that are presented in Kotlin but not in Java. Our goal is to study how Android developers use programming features that were not fully available for developing Android apps before the release of Kotlin.

5.2.1.1 Selection of Kotlin features

To identify Kotlin features that we focus on in this study, we inspected the Kotlin official website. First, we extracted 13 features from a document that compares Kotlin and Java [57]. Then, we extracted 4 features from Kotlin's release notes, which were not mentioned in the comparison document (*coroutine* as experimental feature and *type alias* from release 1.1 and *contract* and *inline class* from release 1.3). Finally, we passed over the Kotlin Reference [129] and identified 7 more features. Table 5.2 summarizes the features identified. Appendix B provides the description and some examples of the use of each feature.

Table 5.2: Kotlin features and their release version.

ID	Feature	Release version	Normalization Criteria
1	Type inference	1.0	# of variable declarations
2	Lambda	1.0	LLOC
3	Inline function	1.0	# of named functions
4	Null-safety (Safe and Unsafe calls)	1.0	LLOC
5	When expressions	1.0	LLOC
6	Function w/arguments with a default value	1.0	# of functions + # of constructors
7	Function w/ named arguments	1.0	# of function calls
8	Smart casts	1.0	LLOC
9	Data classes	1.0	# of classes
10	Range expressions	1.0	LLOC
11	Extension Functions	1.0	# of named functions
12	String template	1.0	# of strings
13	Delegation (Super and Property)	1.0	# of properties # of inheritances
14	Operator Overloading	1.0	# of named functions
15	Singleton	1.0	# of object declarations
16	Companion Object	1.0	# of object declarations
17	Destructuring Declaration	1.0	# of variable declaration
18	Infix function	1.0	# of named functions
19	Tail-recursive function	1.0	# of named functions
20	Sealed class	1.0	# of classes
21	Type aliases	1.1	LLOC
22	Coroutine (experimental)	1.1	LLOC
23	Contract (experimental)	1.3	LLOC
24	Inline class	1.3	# of classes

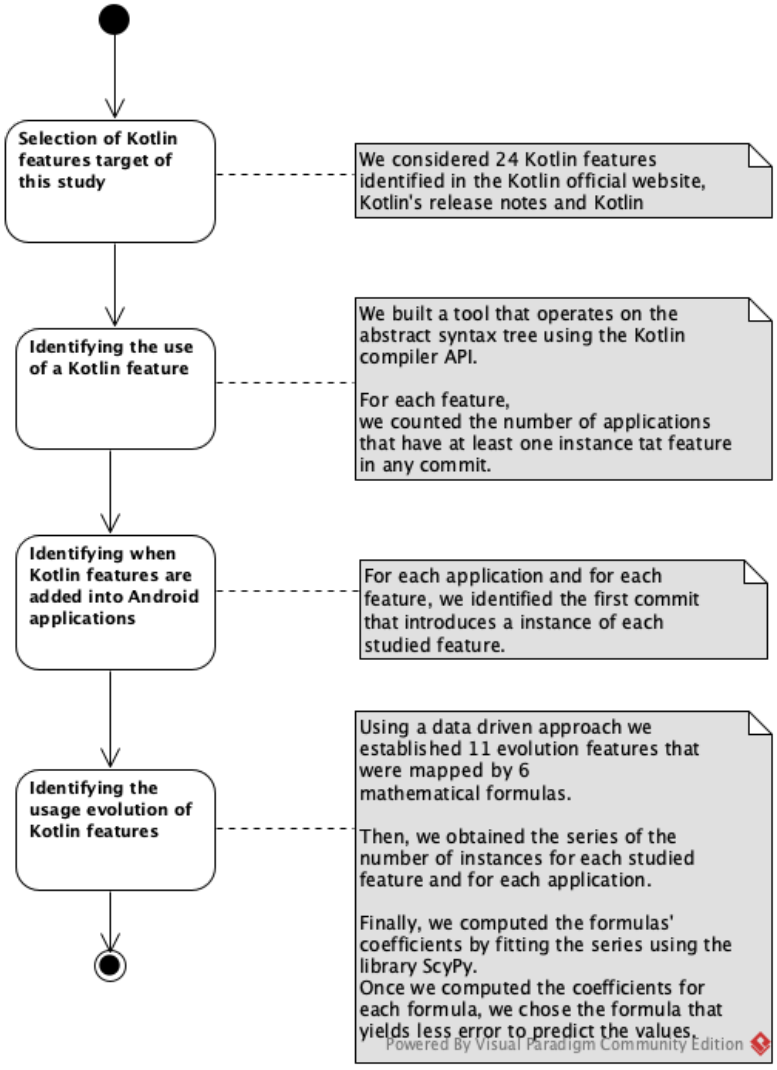


Figure 5.3: Steps executed in out study about the use of Kotlin features.

```
1 val listWithNulls = listOf("Kotlin", null)
2 for (item in listWithNulls) {
3     item?.let { println(it) }
4 }
```

Listing 3: Kotlin snippet with instances of *type inference*, *lambda and safe call*.

5.2.1.2 Identification of Kotlin features

For responding to our research questions, we need to identify the use of Kotlin features from the applications' source code. For example, Listing 3 shows a code that has three Kotlin features from Table 5.2: *type inference* (line 1), *lambda* (lines 2 and 3) and *safe call* (line 3). For that reason, we built a tool that, given as input, a Kotlin source code file (.kt file) returns a list of all features found in that file.

We built the tool as follows. Our feature detection tool operates on the abstract syntax tree (AST) provided by the Kotlin compiler API. For each feature presented in Table 5.2, we manually investigated how a feature is represented on an AST. Then, we encoded different analyzers for detecting feature instances on ASTs. We encoded analyzers successfully for the 24 features presented in Table 5.2. We built 26 analyzers because we encoded two analyzers for two features: *Null-safety* and *Delegation*. We split the *Null-safety* feature in two: *1) Safe call* that provides information about the usage of the safe call operator '?' and *2) Unsafe call* tells whether developers use the not-null assertion operator '!!' that we will refer to as unsafe operator. We also split *Delegation* into two features: *1) Super Delegation* and *2) Property Delegation*. Moreover, regarding the feature *Type inference*, our analyzer focuses on a single scenario: variable declaration (e.g., "var a=10;", the type of *a* is inferred (int)).

To the best of our knowledge, there is no benchmark of Kotlin features usage that we could use for evaluating our tool. Therefore, we performed a study applying manual verification to evaluate its precision. We executed our tool over the last version of each application from FAMAZOA, which returned a list of features instances found, with their respective locations (file name and line number). This information allowed us to verify whether each reported feature instance was present or not in the reported files. To achieve a confidence level of 95% and a confidence interval of 10%, we checked 96 instances of each feature, randomly selected. The evaluation results, available in our online appendix, showed the precision of the tool is 100%.[4][5] To measure our tool's recall, both authors (myself, a Ph.D. student, and my Ph.D. advisors) manually analyzed 100 files randomly selected. Then, we executed our tool over this set of files and calculated the recall. We found a recall of 100% for all features, but *coroutine* (an experimental feature). Our strategy based on keywords could not identify all possible *coroutines*, resulting in a recall of 91%.

To analyze the usage of features along with the history of one application, we created another tool that takes as input a Git repository and produces, for each version v (i.e., commit), the number of features found on v. The tool navigates through the commits of the active branch, in general, the *master* branch. Given a Git repository, it starts from the oldest commit, and for each

[4] https://github.com/UPHF/kotlin_features

[5] To find the minimum number of instances to analyze (96), we compute the confidential level by considering as sample size the number of instances of the most frequent feature, which was *type inference* with 165 667 instances.

commit, it computes the number of features by invoking our feature detection tool described. When a repository is analyzed, our tool generates a JSON file. This file has for each commit, the number of features of each studied feature grouped by file. This tool is built over Coming [182], a framework for navigating Git repositories that allow users to plug-in their source code analyzers.

5.2.1.3 Evaluation dataset

To investigate the adoption and evolution of Kotlin features, we need Android applications written in Kotlin. To collect more applications that satisfy this criterion, we re-execute the methodology presented in Section 3.1.1 to update FAMAZOA, our dataset of open source Android applications written in Kotlin. For that reason, in next studies, we use the third version of FAMAZOA, which contains 387 open source Android applications written fully or partially in Kotlin.[6] Those apps have a median of 6 contributors, 56 files and 3984 lines of Java/Kotlin.

5.2.2 The adoption of Kotlin features

Kotlin provides several programming language features that were not fully available for developing Android apps before its release. In this study, we verify whether Android developers have been using these features by answering the following research question:

- RQ_6: Which Kotlin features are adopted by Android developers?

Section 5.2.2.1 presents the method applied during this empirical study. Section 5.2.2.2 shows the study's results.

5.2.2.1 Analysis method

To answer RQ_6, we processed the output of our feature evolution tool (Section 5.2.1.2). For each feature f, we counted the number of applications that have at least one instance of f in any commit and the total number of *instances* in the last commit of every application.

As the applications may have different sizes, we normalized the number of instances following the criterion presented in Table 5.2. We could normalize each feature with a unique metric of size, such as LLOC. However, we consider it more meaningful (and would better describe the use of a feature) if we normalize each feature f by a metric related to f. For instance, it gives more information to say that an application has 1 data class for each N classes, rather than reporting that it has 1 data class for every Y lines of code.

Now, we detail the normalization process. As explained in Section 5.2.1.2, our analyzer of *type inference* considers only variable declarations. Thus, we normalized the instances of this feature by the number of variables declared. Since *destructuring declarations* break down objects into (declaring) multiple variables, we also normalized its instances by the number of variables declared.

In Kotlin, every declared function is a *named function* node in the AST. Thus, we normalized the number of instances of *extension functions* by the number of *named functions*. Since only *named functions* might receive the modifier *inline*, we used their number to normalize the number of *inline functions*. For the same reason, we used the same criterion to normalize the number of *tail-recursive*. Analogously, we normalized the number of *data classes*, *sealed classes* and *inline class* by the number of classes. Additionally, the number of *named functions* was also used to normalize the number of *operator overloading* because, by definition, an overloaded operator is a *named function* that receives the modifier *operator*.

[6] https://uphf.github.io/FAMAZOA/versions/v3

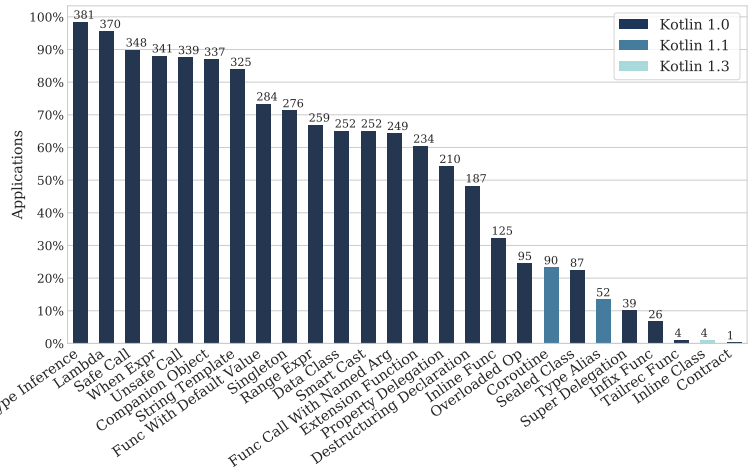


Figure 5.4: Percentage of applications that use a feature. Each bar corresponds to a feature and contains on top the number of applications that use that feature.

Arguments with a default value can be used in *named functions* and *constructors*. Thus, we normalized *function with arguments with a default value* by the number of *named functions* and *constructors*. *Named arguments* are used when a method/function is called. Consequently, we normalized the number of function calls with named arguments by the number of function calls. Besides, since only strings might have a *string template*, we normalized the number of *string templates* by the number of strings. Concerning Kotlin delegations, we normalized the number of properties delegated by the number of properties. Moreover, as *super delegation* is an alternative to inheritance, we normalized their instances by the number of classes. In Kotlin, object expressions are used to declare singletons and companion objects. Consequently, we normalized the number of *singleton* and *companion object* by the number of object declarations. For the remaining features, we normalized them by LLOC because we could not find a better criterion.

5.2.2.2 Results

To answer RQ_6, we applied the method described in Section 5.2.2.1, and Figure 5.4 summarizes its results. For each feature, it shows the percentage of applications that have used that feature at least once (considering all the versions of those apps). Also, Figure 5.5 shows the distribution of the normalized number of occurrences of each feature per application (considering the latest version of each one).

We observed that the most used feature is *type inference*, with 381 out of 387 (98%) applications having at least one *instance* of this feature, as Figure 5.4 shows. Using *type inference*, developers do not need to declare explicitly the types of variables when they are declared. The Kotlin compiler can identify the variable's type by evaluating the expression on the right side of the assign operator '=', see Listing 3 (line 1). In this example, the type of 'listWithNull' is infered as 'List<String?>'. We found that a median of 77% of variable declarations does not explicitly declare their type, as Figure 5.5 displays. However, we noticed some applications whose all variables have their type inferred as well. Therefore, we concluded that developers can write code more concisely, avoiding type declaration when the type is self-explained in the assignment.

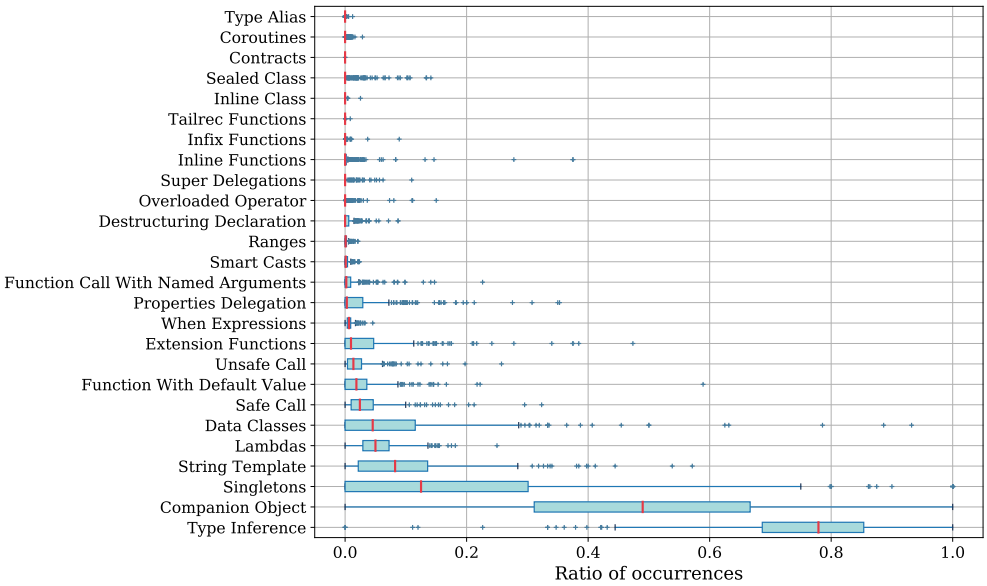


Figure 5.5: Kotlin features normalized.

Lambda is the second most used feature, being found in 370 out of 387 (95%) applications with a median of 71 *instances* per application. The fact that Android applications rely on callbacks to interact with the Android platform [280] could be seen as a reason for the number of instances found. However, a high number of instances, especially when they are nested, could lead developers to write code with poor readability. In Kotlin, developers do not need to name the parameter of a single parameter lambda function. In this case, it is automatically named as 'it'. Therefore, a chain of nested lambdas using this mechanism might be hard to read. For that reason, JetBrains once considered removing the 'it' parameter [276].

Safe call is the third most used feature. We found 348 out of 387 (89%) applications where *safe calls* were found, with a median of 28 occurrences per application. Another feature related to null-safety, *unsafe call*, is used in 339 out of 387 (87%) applications. As the opposite of *safe calls*, the usage *unsafe calls* could result in *NullPointerException (NPE)*. We investigated some instances of this feature manually, and we found cases where it could be substituted by a *safe call* or the Elvis operator.

> **Finding 1:** *We found a median of 16 occurrences of unsafe calls per application, which makes these applications more prone to 'NullPointerException'. However, we found that occurrences could be replaced by Kotlin's built-in functions such as the Elvis operator to avoid NPEs.*

Considering *companion objects*, the sixth most used feature, found in 337 out of 387 (87%) *Companion objects* are the substitute of Java static members in Kotlin, but by default, *companion objects'* properties and methods are not static. To make them static, developers should use the annotation *@JvmField* or *@JvmStatic*, and we found several not annotated objects in our study. As Figure 5.4 shows, *function with arguments with a default value* and *function calls with named arguments* is used in more than 60% applications of our dataset's applications with a median of occurrences of 3 and 2 occurrences per application, respectively. We normalized the number of *instances* by the *number of named functions* plus the *number of constructors* and the *number of*

function calls, respectively, as Figure 5.5 shows.

> **Finding 2:** *According to the Kotlin convention, the use of named arguments can improve the readability of the code [127]. However, we found that less than 1% of function calls have a named argument.*

Moreover, we found that 252 out of 387 (65%) applications using *data classes*, with a median of 1 instance per application. In total, we found that only the 4% of the applications' classes are data classes. Figure 5.5 shows the distribution of that proportion: for 25% of applications, the proportion of data classes is more than 10% of the total number of classes.

Kotlin provides simple approaches to use two well-known design patterns, the *Singleton* and *Delegation*. Regarding the usage of *singleton*, we observed 276 out of 387 (71%) applications have at least one class that implements Singleton pattern. Furthermore, an application has a median of 2 occurrences of this feature.

Considering *properties delegation*, we observed it in 210 out of 387 (54%) applications. Normalizing the number of properties delegated by the number of properties for each application, we found that:

> **Finding 3:** *Although Kotlin standard library provides several useful kinds of delegation, such as lazy and observable, less than 1% of properties defined in Kotlin applications are delegated.*

Coroutine is most used feature considering those released after Kotlin 1.0. Note that *coroutine* was released as an experimental feature in Kotlin 1.1, and it was made stable in Kotlin 1.3 in October of 2019. Comparing with *type alias*, also released in Kotlin 1.1, *coroutines* are found in 23% of applications, whereas *type aliases* are found in 13%. Since in Java, the concept of *type alias* does not exist, *type alias* is not interoperable with Java. Therefore, this could be one possible reason for this level of adoption. On the other hand, *coroutine* might be used to perform different actions concurrently without block Android's main thread. For instance, one could replace *AsyncTaks* usage by *coroutine* since *AsyncTask* was marked as deprecated [70].

Delegation has been proven to be an alternative to inheritance [86], but we found that *super delegation* was used only in 39 out of 387 (10%) applications. Moreover, we observed that Kotlin applications have a median of 0 occurrence of *super delegation*. Therefore, we believe that the fact of inheritance is essentially absent in mobile applications [193] can explain this finding.

Among the least used features, we found two features released in Kotlin 1.3. Although *contract* was released as an experimental feature, we found one application using it, and *inline class* was found in 4 applications.

> **Finding 4:** *Less than 35% of Android apps have instances of features like inline function, inline class and tail-recursive function that might improve their performance [124, 256, 123].*

> Response to RQ_6: *Which Kotlin features are adopted by Android developers?*
> We studied 26 Kotlin features, and as a result, we found that Android developers use all features of them. We identified three groups of features: *i)* 7 features used in at least 80% of applications; *ii)* 9 features used in more than 48% and less than 80% of applications; *iii)* 10 used in less than 33%.
> Furthermore, we found that *type inference*, *lambdas* and *safe calls* are the most used features, being found on 98%, 95% and 89% of applications, respectively.

This result shows that Android developers are taking advantage of using Kotlin features that they could not fully use before Java 7. We found that even experimental Kotlin features, like

Coroutines and Contract, have been used. We also showed that 10 features are used in less than 33% applications. So, we conclude that there is room for improvement in the usage of these features.

5.2.3 The introduction of Kotlin features

Our empirical study about the adoption of Kotlin features showed that Android Developers had used the 26 studied features. In this section, we focus on another aspect of the adoption of these features. We investigate when these features are added into Android applications throughout their evolution by answering the following research question:

- RQ_7: When do Android developers introduce Kotlin features during applications' evolution?

Section 5.2.3.1 presents the method applied during this empirical study. Section 5.2.3.2 shows the study's results.

5.2.3.1 Analysis method

For responding to RQ_7, we computed, for each application a and for each feature f, the first commit C_{fa} that introduces an instance of f into a. Finally, we defined a metric, named *introduction moment*, $m_{af} \in [0, n]$ where n is the number of days between the first Kotlin commit and the last commit, which measures how long after the initial commit a feature f was introduced into a. It is expressed in days. For instance, $m_{af} = 0$ means that feature f was introduced into a in the same day of the first Kotlin commit, $m_{af} = 5$ means that f was added in the 5 days after the Kotlin introduction, and $m_{af} = n$, means that f was introduced in the same day of the last commit.

5.2.3.2 Results

To answer RQ_7, we used a metric defined in Section 5.2.3.1, named *introduction moment*. Figure 5.6 displays its distribution. We found that:

> **Finding 5:** *In the following ten days after the first commit with Kotlin code, 15 out of 26 features were added into Android applications.*

However, comparing the introduction moment of the most and the least used features turned out distinct behaviors. We found that the most used features, *type inference*, *lambda*, *safe call*, *when expressions*, *companion object*, *unsafe call*, *string template and singleton*, presented in median the *introduction moment* smaller than 1. While the least used features presented a high introduction moment.

> **Finding 6:** *The least used Kotlin features tend to be introduced later on the applications' history compared to the most used features.*

Furthermore, the two features with the highest introduction moment, respectively, 308 and 439, were *inline class* and *contract*, both released in Kotlin 1.3. These features were made available more recently compared to the others. This can explain the highest introduction moment. Among features released in Kotlin 1.1, we noted that *coroutine*, released initially as an experimental feature, presented a smaller introduction moment than *type alias*. We analyzed the commits that introduced *coroutine* and found that 87 out of 90 (96%) applications added

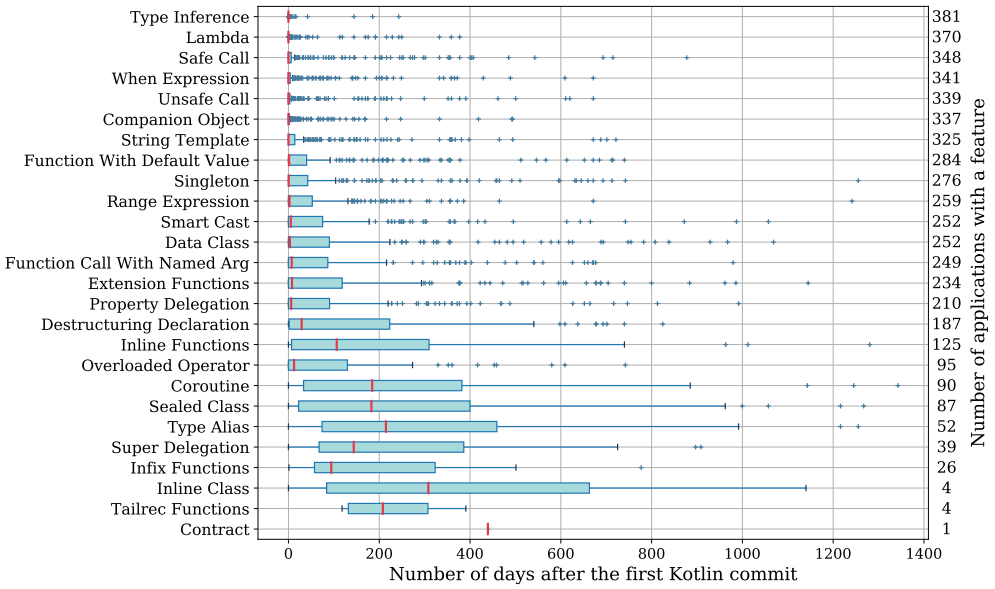


Figure 5.6: Distributions of the number of days between the first Kotlin commit and the commit that introduces the first instance of a feature.

coroutines when *coroutine* was still an experimental feature. *Contract*, found in one application, is another experimental feature used by developers. Therefore, as Dyer [76] found in their study about the usage of Java features, we observed that developers start adopting features early when they are experimental.

> Response to RQ_7: *When do Android developers introduce Kotlin features during applications' evolution?* Most features, 15 out of 26, are added into the first days of development using Kotlin. Moreover, while the most used Kotlin features *type inference*, *lambda*, *safe call*, *when expressions*, *companion object and string template* tend to be introduced in the first commit with Kotlin code, the least used features tend to be introduced later into an application.

These results show that most features are added into applications in the following ten days after the first commit with Kotlin code. Therefore, a solid understanding of these features would help developers use them correctly since the initial development phase.

5.2.4 The usage evolution of Kotlin features

The goal of RQ_8 is to detect trends that describe the usage evolution of features along with the applications' history. For example, we want to detect applications where the use of a particular feature is constant, increases or decreases along with its evolution. Therefore, we conducted an empirical study to answer the following research question:

- RQ_8: How the usage of Kotlin features evolves along with the evolution of Android applications?

To respond to this research question, we mined Android applications written in Kotlin to identify evolution trends. Section 5.2.4.1 presents the evolution trends found in that step.

Section 5.2.4.2 describes the methodology used to identify instances of features throughout applications' evolution, and match features evolution trends with trends considered in this study Section 5.2.4.3 shows the result of this empirical study.

5.2.4.1 Evolution trends

To classify each pair of application-feature (a, f) and find the trend that *best* describes the evolution of a feature f in the history of an application a, we defined a set of evolution trends following a data-driven approach.

First, we automatically plotted a two-dimension plot for each application-feature (a, f) where the axis X corresponded to the number of commits (chronologically ordered) from a, and the axis Y was the number of instances of a feature f. Then, we iteratively analyzed the generated plots. For each of them, we first described the trend we observed in two ways: *1)* natural language description (e.g., "Instances increase"), *2)* mathematically (e.g., "y=ax+b"). Then, we checked whether a similar trend had been seen before based on such descriptions. We repeated this last step until no new trend be found after analyzing dozens of plots. To avoid bias, both authors (myself, a Ph.D. student and my Ph.D. co-advisor) have analyzed the same plots separately, and later, we compared and discussed the result found. Other works have previously defined evolution trends, such as Hecht et al. [115] and Malavolta et al. [169]. Both studies have used their trends to classify evolution plots *manually*. On the contrary, the definition of our evolution trends was motivated by the need to classify trends automatically, which guarantees *i)* scalability (i.e., thousands of apps-feature pairs to classify) and *ii)* the replication (analysis of other applications) of this study.

We found 11 unique evolution trends that were described by 6 different mathematical formulas. An example of each of them is shown in Figure 5.7. These trends are:

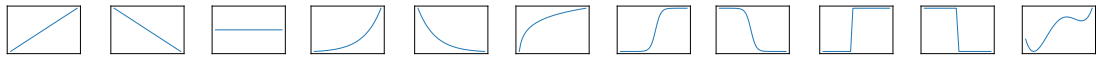
- *Constant Rise (CR)* describes features that, once they are introduced (i.e., used for the first time in an application), developers tend to add more instances of this feature in future application versions. Therefore, the number of instances increases at a constant rate, i.e., linearly along with the application's evolution.

- *Constant Decline (CD)* describes features that, once they are introduced, developers tend to remove them gradually in future applications' versions. Therefore, along with the application's evolution, the number of feature instances decreases at a constant rate.

- *Stability (S)* describes features whose numbers of instances remain the same after their introduction, throughout the application's evolution.

We used the linear function given by the formula $y = ax + b$ to detect CR, CD and S. Since in linear function, the rate of change (given by coefficient a) is always constant, we could classify the application's trend into: (CR) when $a > 0$, which implies on constant increase; (CD) when $a < 0$, which implies on constant decrease; and (S) when $a = 0$, which determines a constant behavior.

- *Sudden Rise (SR)* describes those features that the number of occurrences grows suddenly after relative stability along with the applications' history. Using this trend, we are able to identify those features that present a small number of instances in the first commits and then, at the following commits, on each commit, developers introduce significantly more instances.



(a) Constant Rise. (b) Constant Decline. (c) Stability (d) Sudden Rise. (e) Sudden Decline. (f) Sudden Rise Plateau. (g) Plateau Gradual Rise. (h) Plateau Gradual Decline. (i) Plateau Sudden Rise. (j) Plateau Sudden Decline. (k) Instability.

Figure 5.7: Example of evolution trends. The x-axis shows the evolution of an application, i.e., commits, and the y-axis shows the number of occurrences of a feature.

- *Sudden Decline (SD)*, analogously, describes the opposite behavior of SR, where the number of feature instances decreases suddenly. Features that present many instances since the first commits and then, suddenly, start to be removed and this behavior continues in the next consecutive commits.

We used the exponential function given by formula $y = ab^x + c$ to detect SR and SD. In this formula, the rate of change is given by coefficient b. When b is greater than 1, the value of y increase as x increases. On the other hand, if b is $0 < b < 1$, the value of a function increases as the value of x decreases. Then, considering the value of b, we classified evolution trends better described by an exponential function in two trends, SR and SD.

- *Sudden Rise Plateau (SRP)* describes features which most of its instances are introduced in the firsts commits of an application. Then, during the rest of the application's history, only a few instances are introduced.

We used the logarithmic function given by formula $y = a\log_e(bx) + c$ to detect SRP. In this case, unlike the exponential formula, the rate of change always decreases with time.

- *Plateau Gradual Rise (PGR)* describes those features which once they are introduced, the number of instances tends to remain the same during an interval of commits and then presents a sudden increase in few consecutive commits, and finally presents a stable behavior again.
- *Plateau Gradual Decline (PGD)* is similar to the trend PGR. This trend describes those features that the number of instances starts and finishes stable. However, on the contrary of PGR, here, the change that happens between the periods of stability is a reduction of the number of instances in a few consecutive commits.
- *Plateau Sudden Rise (PSR)* is similar to PGR. However, using this trend, we aim to detect features that present: a) a stability period at the beginning of the application's evolution b) a transition period containing only one commit and c) stability period at the end of the application's evolution.
- *Plateau Sudden Decline (PSD)* analogously to PSR, this trend is a special case of PGD, where the change between the two periods of stability has a marked decrease in the number of instances in two consecutive commits.

We used the Sigmoid function, $y = \frac{L}{1+e^{-k(x-x0)}} + b$, to detect the trends PGR, PGD, PSR and PSD. A Sigmoid is a bounded function where y can assume values from $-L + b$ to $L + b$. Additionally, when $L * b > 0$, y approaches $L + b$ as x approaches $+\infty$ and approaches $-L + b$ as x approaches $-\infty$. On the other hand, we observe the opposite behavior when $L * b < 0$. While $x0$ is Sigmoid's midpoint, which marks the middle of the S-curve, k determines the curve's steepness. The

PGR trend is characterized by the lowest the number of instances ($-L+b$) at the beginning of the application history and the highest value ($L+b$) of the number of instances in the end. Moreover, the transition between the lowest and the highest values is characterized by a gradual rise, in a small number of commits, when developers introduce a considerable number of instances. The PGD trend follows the opposite behavior, starting with the highest number of instances and finishing with the lowest number of instances. The differences between PGR and PSR, and PGD and PSD, is determined by the coefficient k, where a small value of coefficient k makes the slope of a Sigmoid function completely vertical.

Instability (I): describes features that the number of instances alternates between an increase and a decrease during the application's evolution.

We used the polynomial function given by formula $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$, where n is the degree of the polynomial function and $a_n > 0$ to detect the Instability trend. Since the degree of a polynomial determines the number of minimum and maximum (at most $degree - 1$) given an open interval, using this formula, we detected those features which the number of instances throughout the application's evolution form a curve with two or more minimum or maximum.

5.2.4.2 Analysis method

To match a feature evolution trend from one application with one of the studied trends, we applied the following methodology:

Obtaining the series of the number of instances per application. Given one application, our tool's execution gives each feature's number of instances that each application's version (i.e., commit) has. Therefore, for each application a, we generated a series of values $y_{xa} = \{vax_1, vax_2, .., vax_n\}$ where vax_i corresponds to the number of instances of feature f detected in the i-version of a (corresponding to the i-th commit considering chronological order, starting with the first version that introduces x). That is, the first element contains the number of features in the first version that introduces f.

Computing formulas' coefficients by fitting the series. For each series y_{xa} from a pair application-feature (a, x), and for each formula f presented in 5.2.4.1, we used non-linear least squares to fit function f to data y_{xa}. As a result, this gives a set of coefficients $(\alpha_1, .., \alpha_n)$ for f that correspond to the optimal values so that the sum of the squared residuals SS_{reg} (Formula 5.1) is minimized.

$$SS_{\text{res}} = \sum_{i}^{|versions\, a|} (y_{xa_i} - f_i) \tag{5.1}$$

The number of coefficients generated varies according to the formula f: for linear, the number of coefficients is two, whereas a polynomial of 4 degrees is 5. For executing the fitness of data, we used the function 'curve_fit' from library *SciPy* [132].

Post-Processing formulas and coefficients. This step has two goals: *a)* simplify polynomial formulas and *b)* discard some Sigmoid functions which do not have a clear S-shape considering the domain applied $[0, \#commits - 1]$. We simplify those polynomial formulas whose the coefficient a_n is close to zero (i.e., < than 0.0001) because these evolution trends can be similarly described by a polynomial $n-1$ degree. We also discard Sigmoid functions where the coefficient x_0, i.e., the Sigmoid's midpoint, is outside the range of commits $[0, \#commits - 1]$ because in these cases, only one plateau could be in the commit range. Therefore, these cases do not configure any of the studied trends modeled with the Sigmoid function.

Choosing the formula that best represents a feature evolution trend. Once we computed the coefficients for each formula, we chose, for each pair (a, x), the formula that yields less error

to predict the values of y_{xa}. We call it f_{best_ax}. Thus, this formula is considered as the one that best represents the feature evolution trend among all formulas we consider in this study (Section 5.2.4.1). We based our choice on a statistical measure named R-squared (Coefficient of Determination) which measures how close the data are to the fitted formula. R-squared (R^2) is always between 0 (bad) and 1 (good).

When two formulas $f1$ and $f2$, produce the same R^2 values within a small positive delta (0.01 in our study), we prioritized formulas with the smallest quantity of coefficients that is: $Linear >> Exponential >> Logarithmic >> Sigmoid >> Polynomial$.

Summarizing of best formulas. For each feature x and formula f, we count the number of applications that have f as the best formula (Section 5.2.4.2). The result from this step helps us to explain the most frequent evolution trend associated with each feature.

5.2.4.3 Results

Table 5.3 presents the results of RQ_8 obtained using the methodology presented in Section 5.2.4.2. Each cell shows the number and the percentage of applications whose the evolution of a feature f (a row) is *better described* by trend t (a column). Additionally, the number of applications analyzed with a feature f is displayed in the column Total applications.

For 11 out of 26 (42%) features studied are better described by CR, a constant rise trend. Moreover, 7 out of 26 (6%) features are better described by PSR. Other 6 features (23%) presented the behavior of stability intervals separated by a gradual rise (trend PGR Figure 5.7i). We also observed that 2 (7%) features better described by PSD. Note that the feature *tail-recursive function* is better described by two trends, PSR and PSD. Additionally, the PGR trend better describes 1 feature (3%), *inline class*. Finally, we did not find any feature better described by stability.

> **Finding 7:** *In general, the number of instances of features tends to grow along with the applications' evolution.*

In Table 5.3, the column **Inc** shows the total number of applications and the percentage of applications better described by any trend whose the number of instances increases throughout the application's evolution (i.e., CR, SR, SRP, PGR, PSR). Analogously, the column **Dec**, shows the sum of CD, SD, PGD, PSD. Furthermore, column **I** represents the Instability trend. Consequently, 100% of applications are represented by columns **I**, **Inc** and **Dec**.

Now we explain our results for the three most used Kotlin features, *type inference*, *lambda*, and *safe call*. We found that most applications presented a behavior of increasing the number of instances throughout the applications' evolution. Table 5.3 shows this behavior in 72% of applications containing *type inference*, 79% of applications containing *lambda* and 79% of applications containing *safe calls*. Moreover, the constant rise trend, CR, better described the evolution of these features in 36%, 36% and 30% of applications, respectively. Considering the usage of *type inference*, we found 91 (26%) applications whose number of instances varies between increase and decrease intervals along with applications' evolution. Furthermore, we observed only 8 (2%) applications whose number of *type inference* instances decreases during applications' evolution.

Finally, as we described in Section 5.2.4.1, to identify the trend that better describes the evolution trend of a feature, we used R-squared, which measures how close the data are to the fitted formula. The R-squared assumes values between 0 and 1, and 1 means a perfect fitting. In our study, the median R-squared was 0.88, which means that 50% of the selected formulas describe almost perfectly the evolution trends. Moreover, we found that 75% of evolution trends fitted presented an R-squared greater than 0.74, and only outliers have R-squared values lower than 0.43.

Table 5.3: This table shows for each Kotlin feature X (rows) and for each evolution formula f (columns) the number of applications where the formula f better *describes* the evolution of the feature X. The last row, Total, shows how many features were better described by a trend (column)

Features \ Trends	CR	CD	S	SR	SD	SRP	PGR	PGD	PSR	PSD	I	Inc	Dec	Total
Type Inference	**128 (36%)**	3 (1%)	0 (0%)	60 (17%)	1 (0%)	53 (15%)	10 (3%)	0 (0%)	5 (1%)	4 (1%)	91 (26%)	256 (72%)	8 (2%)	355
Lambda	**124 (36%)**	2 (1%)	0 (0%)	67 (20%)	4 (1%)	31 (9%)	36 (10%)	0 (0%)	14 (4%)	4 (1%)	61 (18%)	272 (79%)	10 (3%)	343
Safe Call	**94 (30%)**	2 (1%)	0 (0%)	64 (20%)	3 (1%)	38 (12%)	33 (10%)	1 (0%)	21 (7%)	9 (3%)	51 (16%)	250 (79%)	15 (5%)	316
When Expr	**85 (29%)**	3 (1%)	0 (0%)	47 (16%)	2 (1%)	28 (10%)	72 (24%)	0 (0%)	29 (10%)	5 (2%)	23 (8%)	261 (89%)	10 (3%)	294
Unsafe Call	**60 (19%)**	32 (10%)	0 (0%)	54 (17%)	19 (6%)	25 (8%)	41 (13%)	3 (1%)	17 (5%)	11 (4%)	51 (16%)	197 (63%)	65 (21%)	313
Companion Object	**75 (26%)**	11 (4%)	0 (0%)	38 (13%)	8 (3%)	33 (11%)	57 (20%)	2 (1%)	39 (13%)	5 (2%)	22 (8%)	242 (83%)	26 (9%)	290
String Template	**80 (27%)**	5 (2%)	0 (0%)	39 (13%)	6 (2%)	32 (11%)	59 (20%)	1 (0%)	28 (9%)	7 (2%)	38 (13%)	238 (81%)	19 (6%)	295
Func With Default Value	50 (21%)	3 (1%)	0 (0%)	41 (18%)	4 (2%)	19 (8%)	**64 (27%)**	3 (1%)	27 (12%)	6 (3%)	17 (7%)	201 (86%)	16 (7%)	234
Singleton	33 (15%)	6 (3%)	0 (0%)	36 (16%)	7 (3%)	25 (11%)	**51 (22%)**	3 (1%)	29 (13%)	24 (11%)	13 (6%)	174 (77%)	40 (18%)	227
Range Expr	30 (14%)	14 (7%)	0 (0%)	28 (13%)	5 (2%)	17 (8%)	**47 (22%)**	6 (3%)	38 (18%)	14 (7%)	12 (6%)	160 (76%)	39 (18%)	211
Smart Cast	38 (18%)	14 (7%)	0 (0%)	35 (17%)	8 (4%)	20 (10%)	**40 (19%)**	4 (2%)	25 (12%)	14 (7%)	10 (5%)	158 (76%)	40 (19%)	208
Data Class	44 (22%)	2 (1%)	0 (0%)	28 (14%)	5 (2%)	19 (9%)	**51 (25%)**	0 (0%)	35 (17%)	6 (3%)	14 (7%)	177 (87%)	13 (6%)	204
Func Call With Named Arg	**48 (22%)**	2 (1%)	0 (0%)	40 (19%)	6 (3%)	20 (9%)	41 (19%)	1 (0%)	30 (14%)	8 (4%)	19 (9%)	179 (83%)	17 (8%)	215
Extension Function	**47 (23%)**	4 (2%)	0 (0%)	32 (16%)	7 (3%)	23 (11%)	42 (21%)	1 (0%)	22 (11%)	7 (3%)	16 (8%)	166 (83%)	19 (9%)	201
Property Delegation	**40 (22%)**	11 (6%)	0 (0%)	32 (18%)	8 (4%)	22 (12%)	23 (13%)	3 (2%)	18 (10%)	5 (3%)	17 (9%)	135 (75%)	27 (15%)	179
Destructuring Declaration	23 (16%)	9 (6%)	0 (0%)	14 (10%)	5 (3%)	9 (6%)	**33 (23%)**	4 (3%)	25 (17%)	16 (11%)	5 (3%)	104 (73%)	34 (24%)	143
Inline Func	11 (12%)	5 (5%)	0 (0%)	11 (12%)	5 (5%)	5 (5%)	17 (18%)	2 (2%)	**20 (21%)**	14 (15%)	5 (5%)	64 (67%)	26 (27%)	95
Overloaded Op	9 (14%)	3 (5%)	0 (0%)	5 (8%)	2 (3%)	5 (8%)	11 (17%)	2 (3%)	**15 (23%)**	11 (17%)	1 (2%)	45 (70%)	18 (28%)	64
Coroutine	**17 (22%)**	1 (1%)	0 (0%)	12 (16%)	1 (1%)	8 (11%)	16 (21%)	0 (0%)	6 (8%)	1 (1%)	14 (18%)	59 (78%)	3 (4%)	76
Sealed Class	6 (9%)	3 (5%)	0 (0%)	10 (15%)	1 (2%)	5 (8%)	7 (11%)	0 (0%)	**26 (40%)**	3 (5%)	4 (6%)	54 (83%)	7 (11%)	65
Type Alias	2 (7%)	1 (3%)	0 (0%)	6 (20%)	1 (3%)	0 (0%)	8 (27%)	0 (0%)	**10 (33%)**	2 (7%)	0 (0%)	26 (87%)	4 (13%)	30
Super Delegation	3 (10%)	0 (0%)	0 (0%)	1 (3%)	3 (10%)	4 (14%)	3 (10%)	0 (0%)	**10 (34%)**	4 (14%)	1 (3%)	21 (72%)	7 (24%)	29
Infix Func	3 (19%)	0 (0%)	0 (0%)	0 (0%)	1 (6%)	0 (0%)	3 (19%)	0 (0%)	3 (19%)	**6 (38%)**	0 (0%)	9 (56%)	7 (44%)	16
Inline Klass	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	**1 (100%)**	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (100%)	0 (0%)	1
Tailrec Func	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	**1 (50%)**	**1 (50%)**	0 (0%)	1 (50%)	1 (50%)	2
Contract	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	**1 (100%)**	0 (0%)	0 (0%)	1 (100%)	0 (0%)	1
Total	11	0	0	0	0	1	6	0	7	2	0	-	-	-

Response to RQ 4: *How the usage of Kotlin features evolves along with the evolution of Android applications?* Developers tend to add more instances along the evolution of Android applications of 24 out of 26 (92%) features studied.

These results show that developers add more instances along with the applications' evolution, which could be a consequence of the fact that developers are more likely to introduce new classes rather than adding code to existing ones [94]. As the number of instances grows, applications become more prone to overusing of features.

5.2.5 Discussion

As a consequence of our study, we defined several implications that we group according to the needs of researchers, tool builders, and software developers.

5.2.5.1 Researchers

The findings from RQ_6 reveal some novel research opportunities:

- *Lambda* is the second most used feature and with a median of 71 instances per application. This finding reveals that Android developers rely on *lambda* to build their applications. Mazinanian et al. [186] studied the usage of *lambda* in Java and found several misuse cases. Therefore, one could perform a similar study considering the use of *lambda* in Kotlin. Another possibility is to investigate how different is the usage of *lambda* in mobile applications compared with non-mobile applications written in Java and Kotlin.

- *Safe call* and *Unsafe call* are used in the majority of applications written in Kotlin. While *safe call* can protect applications of *NullPointerException (NPE)*, *unsafe call* makes them more vulnerable. The study performed by Coelho et al. [55] have shown that considering uncaught exceptions caused by errors in programming logic in Android applications, the NPE is the most common exception. Therefore, researchers can further explore if the usage of *safe call* prevents the occurrence of these exceptions effectively and, on the contrary, if the usage *unsafe call* impact negatively.

5.2.5.2 Tool builders and IDE designers

Android developers rely on lambdas expressions, which might result in a more concise code than the approach that uses inner classes. However, a large number of instances can result in overuse. Therefore, IDE can warn developers about overused lambda, for instance, when a high number of lambdas are nested.

We found a median of 16 occurrences of *unsafe calls* per application that includes some situations where developers could use Kotlin's built-in functions instead. Considering that removing unnecessary instances of *unsafe calls* could improve the null-safety of Android applications, automated code assistance could suggest this type of refactoring.

RQ_6 highlights that less than 35% of Android applications have instances of *inline function*, *inline class* and *tail recursive*, although accurate use of them could improve applications' performance. Therefore, IDEs could detect refactoring scenarios and recommend more efficient implementations.

5.2.5.3 Developers

Regarding the use of *named arguments*, we consider that developers could write a more idiomatic code following the Kotlin conventions that suggest using named argument when a method takes multiple parameters of the same primitive type or for parameters of Boolean type [127]. IDEs could identify spots where a *named argument* could be applied to follow the Kotlin conventions. On the contrary, we noted that Android Studio renders the name of arguments in function calls when they are not provided, resulting in more readable code. However, when someone tries to read that code outside the IDEs, for instance, to perform a code review on GitHub, the readability is not the same as in the IDE.

We observed that less than 1% of properties defined in Kotlin applications are delegated. After inspecting the applications' code, we found opportunities to use property delegation to make a more concise code. As suggested in the official Android development documentation, property delegation could save development time in some scenarios [73]. Better tutorials and training materials could help to promote this feature.

In RQ_7, we found that most features are added into applications in the following ten days after the first commit with Kotlin code. As pointed by Tufano et al. [263], most of the times, code artifacts are affected by bad smells since their creation. Therefore, a solid understanding of these features would help developers use them correctly since the initial development phase, potentially improving the code quality. A dedicated training focused on these features would especially help novice Kotlin developers. On the contrary, the least used Kotlin features tend to be introduced later than the most used features. A possible reason is that developers need more experience with Kotlin to start using these features. Promoting these features could help developers identify opportunities for using them in the early days of development.

We observed that developers add more instances along with the applications' evolution. As the number of instances grows, applications become more prone to overusing of features. Continuous integration and static analyzers could track the use of features and identify misuse and anti-patterns.

5.2.6 Threats to Validity

5.2.6.1 Internal

Selection of features. We investigated the adoption, and usage evolution of a subset of Kotlin features not available in Java. As the criterion used to select the targeted features impacts our results, we consider the ones listed in the official Kotlin documentation to avoid bias [57].

Feature identification. Our results depend on how precisely we identify the target features. Due to the absence of a benchmark of the use of Kotlin features, we evaluate our tool's performance by manually analyzing 96 instances of each feature, and we found 100% precision.

Evolution trend identification. One of the goals of our work is to identify evolution trends. To avoid any bias, we established 11 evolution trends (See Section 5.1.2) similarly used in other studies [99, 115, 169]. Moreover, to match a feature evolution with one of the studied trends, we defined 6 formulas. However, it is possible that other formulas not used in this thesis fit better with any evolution trend.

5.2.6.2 External

Representativeness of FAMAZOA. Our work relies on FAMAZOA [99], a dataset of open source mobile applications written in Kotlin. Considering the number of applications published on Google Play, FAMAZOA represents a small parcel since it only contains open source applications,

limiting the generalization of our findings. However, to the best of our knowledge, it is the largest dataset of Android open source applications written in Kotlin.

Developer's experience. In our work, we analyzed the source code of different Android applications. The use of some studied features may require more experience from developers to be used properly. However, we did not consider information about developers' experience. Nevertheless, all applications analyzed were published on F-droid or Google Play, then it represents a current snapshot of Android development. We will consider this aspect in future work.

5.3 Summary

This chapter presented four empirical studies focused on understanding how Kotlin code evolves throughout Applications' evolution. In the first study, we analyzed how the amount of Kotlin code changes over Android applications' evolution. In the second study, we investigated different aspects related to the usage of Kotlin features by Android developers: *i)* **which** features are adopted, *ii)* **what** is the degree of adoption, *iii)* **when** are these features added into Android applications for the first time, and *iv)* **how** the usage of features evolves along with applications' evolution.

To perform these studies, we mined applications from FAMAZOA, the largest publicly available dataset of open source Android applications written in Kotlin. In the first empirical study, we navigated through each application's commits to measure the amount of Java and Kotlin code on each application. Analyzing this information, we identified 12 code evolution trends. We manually classified the trend that better described the evolution of the amount of Kotlin code on each application. In the second study, we analyzed the usage of 26 Kotlin features. We navigated through all commits of each application to identify the use of these features. Moreover, to classify their evolution trends, we defined 11 evolution trends modeled by 5 functions. Using these functions, we were able to classify each feature trend automatically.

The result of the first study showed that for the 63.9% of applications that include Kotlin code, the amount of code written in that language increases throughout the application evolution. At the same time, the amount of Java code decreases or remains constant. Furthermore, 25% of Android applications have been totally migrated to Kotlin. The second study showed that the majority of Android applications use 15 out of 26 studied features. *Type inference*, *lambda* and *safe call* are the most used features. They were found in 98%, 95% and 89% of applications, respectively. Furthermore, we observed that the most used features are added in the first commit with Kotlin, and the least used features are introduced in the latest commits. Finally, we found that 24 out of 26 features presented an increase of feature instances throughout applications' evolution.

This work allowed us to conclude that it tends to become the dominant language once Kotlin is added to Android applications. Moreover, we can affirm that a significant portion (25%) of applications migrated to Kotlin (Section 5.1.4), even though there is no scientific evidence about the advantages of using Kotlin to create Android applications [59]. Chapter 6 investigates the feasibility of using a machine learning model to assist these developers who want to migrate their applications.

Chapter 6

Applying machine learning to assist the migration of Android applications

In 2019, Google announced that Android became 'Kotlin-first', which means that new APIs, libraries and documentation will initially target Kotlin programming language [277]. Furthermore, Google has started to advise developers to create new applications using Kotlin instead of Java [69]. These facts show that Google is "putting pressure" on developers to move to Kotlin [150]. Fortunately, Android developers, who have their apps already written in Java, do not need to migrate their applications completely because Kotlin is interoperable with Java (See Section 2.1 about language interoperability). Thus, an application could have some files written in Java and others written in Kotlin, all working together. As shown in Chapter 3, Android developers have taken advantage of this characteristic. We found that 162 out of 244 (66.39%) open source Android applications containing Kotlin code also have Java code in their last version.

Our previous study presented in Section 5.1 shows that 25% of the open source applications studied have migrated entirely from Java to Kotlin. Nevertheless, this phenomenon does not only happen in open source applications. For instance, Duolingo is an example of a commercial application that was migrated from Java to Kotlin. Duolingo is a free science-based language education platform that has become the most popular way to learn languages online with 300 million users [227]. The first version of the Duolingo application for the Android platform was written in Java. After five years of existence, all that time using Java for Android, the company that develops the Duolingo app, Duolingo Inc, decided to migrate its Android application from Java to Kotlin. The migration process started in 2017 and it took 2 years. Figure 6.1 shows the evolution of the migration process reported by Duolingo's company.[1] Axis Y shows the amount of code (LOC) written, and axis X the time from 2014. After the introduction of Kotlin, early 2018, the amount of Kotlin code (in green) has increased, and, at the same time, the amount of Java code decreased. In Chapter 5, we identified this trend, and we showed that 19.7% of 244 open source applications analyzed followed that trend.

According to Duolingo's team, the migration to Kotlin was a success in terms of developers' productivity and happiness [45]. However, this case shows that migration from Java to Kotlin is a challenging and time-consuming task (2 years), even for a big company which could have enough resources and developers to carry out the migration activity. Therefore, smaller companies or single developers who want to migrate their applications would probably face a similar challenge

[1] The figure is that one posted on Duolingo's blog [227].

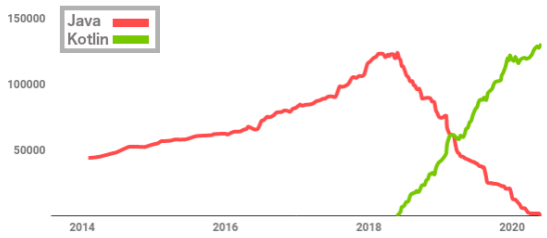


Figure 6.1: Evolution of the number of lines of Java and Kotlin along with the Duolingo application's migration process adapted from [45]. Axis Y shows the amount of code (LOC) written, and axis X the time from 2014.

.

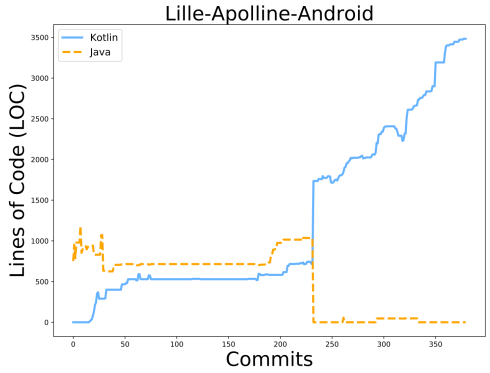


Figure 6.2: Evolution of amount of Java and Kotlin code (LOC) from real open source application that was migrated completely in one single commit.

with fewer resources (i.e., developers) than a big company such as Duolingo.

To speed up the process of migration, developers may use the official Android IDE, which provides the functionality to convert a Java file automatically to Kotlin. Using that feature, the developers could migrate their applications completely to Kotlin with few clicks. Therefore, it is possible to convert a project entirely with several clicks. Applying this approach, developers may perform what we call in the thesis, *one-step* migration. We define *one-step* migration as a migration process that happens in one single commit, as shown in Figure 6.2. However, we believe that migration in one-step is not a feasible approach for big projects because each migrated piece of code (e.g., a class) must be *exhaustively* tested after the migration to verify whether it preserves the expected behavior.

A gradual migration means that a Java-based application is converted to Kotlin throughout its versions (commits). Thus, an application migrated gradually has at least three versions considering the presence of Java and Kotlin: 1) one or more versions containing only Java code, 2) one or more versions containing Java and Kotlin code, and 3) one or more versions containing Kotlin code. As plot 6.1 shows, Duolingo has migrated its application by applying a gradual migration. A gradual migration allows teams to prioritize part of the application to be migrated based on different criteria. For example, one team could migrate the largest files or any other criteria initially according to the team's experience with Kotlin. Moreover, as Duolingo reports [46], it allows applying strict testing, code review and code style of each part of the

migrated application.

Since Android developers have a possibility to migrate partially their applications thanks to the interoperability between Java and Kotlin (See Section 2.1), to start a gradual migration, developers have to decide which file(s) should be first migrated. However, beyond some advice made by Google [64], to the best of our knowledge, there is no tool available or study in the literature that helps developers decide which file to migrate first. Such a tool could identify files more adequate for migration and, combined with the auto-converter from Android Studio, could accelerate the process of migration.

In this thesis, we propose an approach to support developers in the gradual migration of their applications. Given an application, our approach suggests the files to migrate. Suppose that a developer wants to migrate his application completely. Using the proposed approach, a gradual migration will be fostered through file(s) recommendations that it would be more *convenient* to migrate given the current state of that application. The approach decides that based on different metrics.

Our approach works as follows: given as input an application written in Java (and potentially in Kotlin as well) that should be migrated to Kotlin, it first extracts information from each application's file. This information consists of metrics extracted from the source code. For example, one metric is Sources Line Of Code (SLOC), which measures the size of a file by counting the source code's number of lines. Finally, as output, it recommends a list of files to be migrated in order to carry out a gradual migration. The recommendations are performed based on the analysis of the extracted features. We consider that a tool that supports developers choose from which files to start the migration could help developers in the migration process.

We build our approach applying different machine learning techniques to create a model based on migration performed by developers in open source projects (which includes Android applications and other types of Java-Kotlin applications such as a back-end app). Our intuition is that using information from these migrations, our model captures the rationale behind these migrations. For instance, developers may start migrating test file or may prefer migrating the most smelly files or files involved in bugfix activities. Thus, we intend that the learned model capture that knowledge. Other works have used machine learning on other software engineering problems such as code smell detection [85, 8, 18, 74], refactoring prediction [15], change prediction [44] and reliability prediction [242]. However, to the best of our knowledge, nobody has applied machine learning in the context of Android application migration.

In this chapter, we present a study investigating the feasibility of our approach to recommend file-level migrations of Android applications by using two machine learning techniques: *classification* (to predict the files that should be migrated) and *learning-to-rank* (to rank files where top files should be migrated first). We address the following research questions:

- RQ_9: To what extent a classification model learned from migrations done in real projects may recommend migration of files precisely?

- RQ_{10}: To what extent a learning-to-rank model learned from migrations done in real projects may recommend migration of files precisely?

To carry out our study, first, we built a large-scale corpus of open source projects that migrated Java files to Kotlin. Then, we trained different machine learning models using the information extracted from these projects. To validate our models, we verify whether these recommendations made by them correspond to real migrations.

We consider that this initial model is an initial step towards a fully automated recommendation system that supports applications' migration. We believe that in the future, we can integrate

this model with the official Android IDE and, consequently, reducing the time and cost needed to migrate an app.

The chapter continues as follows. Section 6.1 presents an overview of our approach, the objects that our study targets and the methodology applied to extract from these objects the features used by our approach. Section 6.2 describes the empirical study that we perform to evaluate our approach's feasibility using machine learning classification models. Section 6.3 describes the empirical study that we perform to evaluate our approach's feasibility using machine learning learning-to-rank models. Section 6.4 presents a discussion about the result of these empirical studies. Section 6.5 reports the threats to validity. Finally, Section 6.6 summarizes our work and outlines further works to perform on this research topic.

6.1 Study design

We vision an approach capable of recommending candidate files to be migrated during a gradual migration. The recommendation is based on several source code metrics like inheritance metrics, communication metrics and complexity metrics, which have been used in a wide variety of applications and empirical studies related to source code in general (i.e., fault prediction, testing, refactoring) to assess the overall quality of the software [138, 247, 80, 241, 113]. Therefore, the study aims to evaluate the feasibility of building a recommendation system based on supervised machine learning and developers experience to suggest file migrations. For that reason, we explore the possibility of using information from projects that have done file migrations, i.e., developers' experience, to train a machine to learn models that recommend files to be migrated.

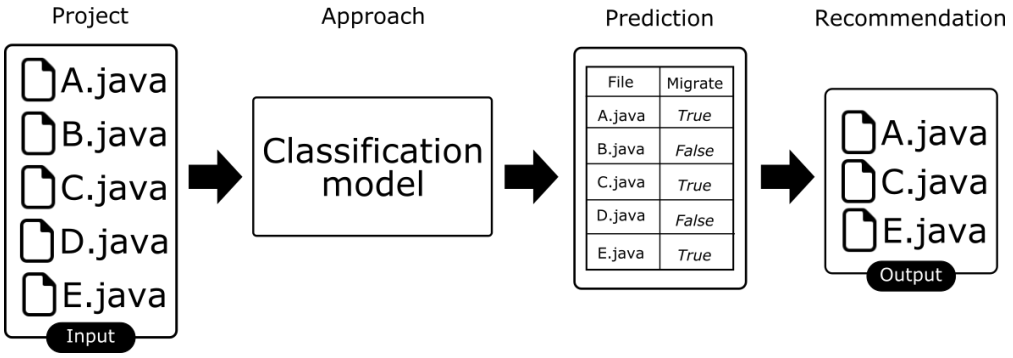
In this section, we describe the main aspects considered to perform a study to investigate that recommendation system's feasibility. In Section 6.1.1, we present our approach overview. In Section 6.1.2, we describe the sets of applications used to train and evaluate our empirical studies' models. Section 6.1.3 describes the features (code metrics) considered by our models.

6.1.1 Approach

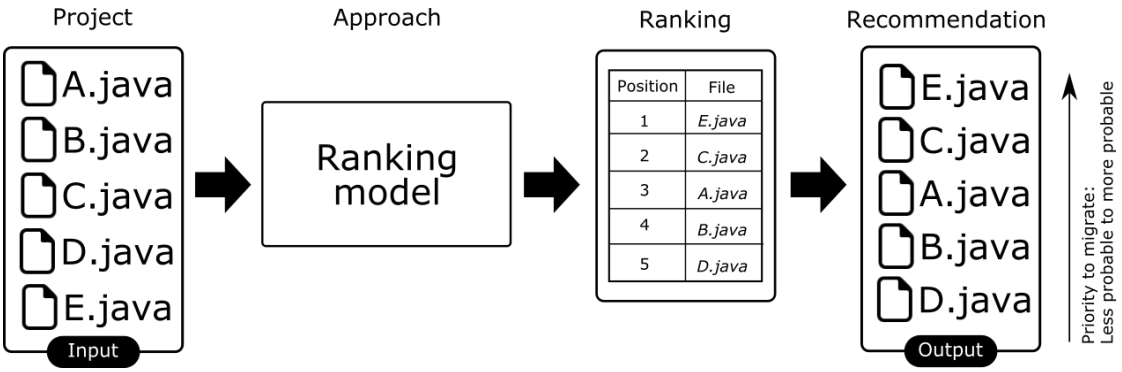
In this section, we first introduce the overall process of our approach. Then, we explain its internal steps.

Suppose that a company wants to migrate a project P written using a programming language $lang_1$ to a second programming language $lang_2$. Thus, this project is the input of our approach. Our approach extracts information from every file written in $lang_1$. Then, using this information, we feed a machine learning model, which was trained with information extracted from projects that have performed migrations. Afterward, a list of candidate files to be migrated is produced as output. We argue that this approach can be applied to the migration between any two languages. Considering the context of Android application migrations, with the list of candidate files to be migrated, developers can manually migrate those files in the recommendation list or even use the auto-converter tool available in Android Studio.

In this work, we apply two machine learning techniques to evaluate the feasibility of our approach: i) Classification: classification models attempt to predict files that should be migrated and ii) Learning-to-rank: learning-to-rank models produce a ranking of files where top files should be migrated first. To clarify the differences between the two techniques, consider the hypothetical situation illustrated in Figure 6.3, where we apply our approach to a Java project with five files. We can apply classification as Figure 6.3a shows to predict the files that should be migrated. In that example, our approach predicted that 3 out of 5 files should be migrated. Therefore, our approach outputs a list containing these 3 files. On the other hand, when a



(a) Applying classification, our approach predicts files that should be migrated. Therefore, the recommendation (output) is the set of files which a model predicted as migrated.



(b) Applying classification, our approach ranks all project files. Then, it recommends the migration of these files according to the files' position in the ranking.

Figure 6.3: This figure shows how our approach applies classification and learning-to-raking to recommend file-level migrations. In this example, a given project has five files and according to the technique applied our approach suggests a set of files to be migrated.

learning-to-rank technique is applied to that example, it generates a list containing every project file, as Figure 6.3b shows. These files are ranked according to their priority to be migrated, i.e., the top files should be migrated first. Considering that the example project has only 5 files, both techniques seem appropriate to that problem, although their strategies are different. In classification, the decision to recommend a file is based individually on that file. On the other hand, the decision taken using a learning-to-rank implementation considers the totality of files.

Independently of the machine learning technique selected, our approach consists of two phases: the development phase and the serving phase. In the development phase, our approach learns a model from the ground truth (i.e., knowledge on files migrated in real projects). Then, in the serving phase, given a project P with code written in $lang_1$ the model generated in the development phase is used to recommend file-level migrations.

6.1.1.1 Development phase

Figure 6.4 shows all steps of this phase. To build our model, it is necessary a ground truth of knowledge about migrations. Therefore, in this phase, we need to analyze projects which have

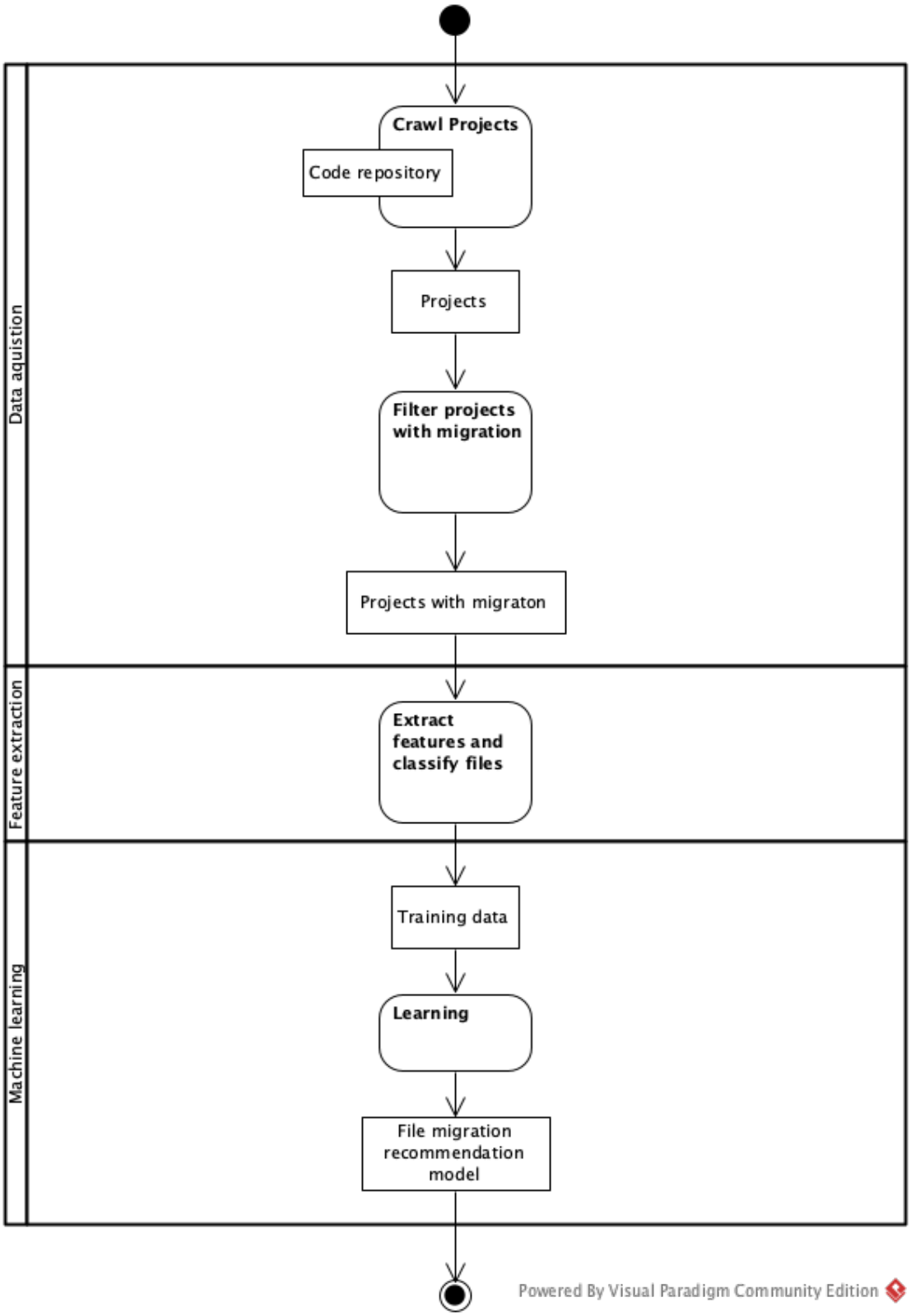


Figure 6.4: In the development phase, we first collect several projects. Then, we filter projects that have migration. Later, we extract features from files and classify files between migrated or not migrated. Finally, using these information we train a model using projects with migration.

performed file migrations from $lang_1$ to $lang_2$. Consequently, this phase's first step is to crawl projects' repositories, as Figure 6.4 illustrates. Then, we analyze these projects to identify those that contain migrations. Section 6.1.2 describes precisely the strategy used to identify these projects.

Once we identified all projects with migrations, we navigate through every project's commit to label their files between migrated or not migrated. We use a heuristic to determine files that are migrated. For example, if a commit removes a file named *A.lang1* and adds another *A.lang2*, we label *A.lang1* as 'migrated'. On the contrary, if the commit updates *B.lang1* then this file is labeled as not migrated. We also extract several source code metrics that we represent as a vector, V, containing a set of feature values extracted during this analysis. The details about the features extracted and the method used to extract such features are explained in Section 6.1.3. The feature vector can be represented as $V = (f_1, ..., f_i, ..., f_n)$, where f_i denotes the value of the i_{th} feature, and n denotes the total number of features. These vectors are part of the training data used by our approach to learning a model. Finally, at the end of this process, the learned model is deployed and ready to be used in the serving phase.

6.1.1.2 Serving phase

In the serving phase, our approach takes as input a program P, written partially or totally using $lang_1$, which developers want to migrate to $lang_2$. As done in the development phase, our approach extracts all metrics from the project's files, i.e., candidates files to migrate, and similarly, it creates for each file the vector of features. These vectors are given as input of our model. Finally, using this information, it outputs a list of recommended files to be migrated.

6.1.2 Projects analyzed in the study

To recommend file migrations, our approach relies on developers' experience with migrations. As our approach is built using a supervised machine learning algorithm, it is necessary to provide data to train our model. Our model works at the level of file-level migrations, but to the best of our knowledge, there is no dataset of file migration from Java to Kotlin. Initially, we would use our dataset named FAMAZOA (Section 3.1.3) to train our models, but it contains only 387 applications, and just a portion of 69% of them have migrations. We detail how we found this number in Section 6.1.2.2. As we want to work on large and heterogeneous data, we decide to create another dataset not limited to Android applications, which was the focus of our first studies presented in Sections 3.1, 3.2, 4.2, 4.3, 5.2 and 5.1. Therefore, we conduct our empirical studies using two datasets: i) a dataset of open source projects with migrations from Java to Kotlin, and ii) a dataset of open source Android projects with migrations from Java to Kotlin.

6.1.2.1 $GitHub_{j2k}$: dataset of open source projects with migrations

We followed 3 steps to create our $GitHub_{j2k}$ dataset: i) identification of open source projects hosted on GitHub that use Kotlin, ii) filtering projects that have Java code at any version, i.e., commits, and iii) filtering projects that have migrated files from Java to Kotlin.

Step 1. Identification of open source projects written in Kotlin hosted on GitHub. This step aims at finding all repositories on GitHub potentially written in Kotlin. Our search was performed on the publicly-available GitHub mirror available on Google BigQuery [119].[2] This mirror offers a full snapshot of the content of more than 2.8 million open source repositories and almost 2 billion files. Moreover, it provides information about the use of programming languages

[2] We queried the GitHub mirror at 23 of January of 2020.

in last commit of each repository. Therefore, we performed a query looking for projects that have Kotlin. As a result, it returned 7 119 repositories.[3]

Step 2. Identification of projects that used Java at its life-cycle. The previous step is necessary to identify projects that have Kotlin. Moreover, we need to filter projects that contain Java as well, since this is a requirement to have migrations. For that reason, we select all projects with at least one commit with Java (i.e., a commit that introduces Java code). At the end of this procedure, we identified 5 126 repositories.

Step 3. Identification of file migration. In order to find real cases of migrations, we navigate through all commits of 5 126 repositories identified in step 2. Then, we apply the following procedure: consider that a repository is a set of versions (commits) $C_r = \{c_i, c_{i+1}, ..., c_n\}$ where i determines the commit number, i.e, c_1 is the first commit and c_n is the last commit. Then, to find migrated files, we compare consecutive commits, c_i, c_{i+1} to extract a pair of files, f_i, f_{i+1}, that should respect the following conditions: i) f_i is a Java file from c_i and it was removed in c_{i+1}, ii) f_{i+1} is a Kotlin file added on c_{i+1}, and iii) f_i and f_{i+1} share the same filename ignoring the file extension (.java, .kt). In this step, we stop once a migration is found. The rationale behind this step, is to keep only repositories with migrations, to save computation time and storage resources. Applying this strategy, we identified 1 357 repositories with migrations.

6.1.2.2 $Android_{j2k}$: dataset of Android applications with migrations

To build our dataset of Android applications with migrations, we mined the repositories of FAMAZOA v3. FAMAZOA is the largest publicly available dataset of open source applications written in Kotlin, and it contains 387 applications written partially or totally in Kotlin.[4]

We applied steps 2 and 3 presented in Section 6.1.2.1, and we identified 270 out of 387 (69%) applications with at least one migration from Java to Kotlin. Since FAMAZOA includes applications hosted on the GitHub, to avoid duplicates, we removed 170 applications from $GitHub_{j2k}$ that are present in $Android_{j2k}$. We ended with 270 applications in $Android_{j2k}$ and 1187 projects in $GitHub_{j2k}$.

6.1.3 Feature extraction

To the best of our knowledge, there is no study that establishes a relationship between any metric and source code migration. Therefore, we consider several source code metrics that have been proven to be useful in other prediction models in software engineering [249, 15]. Moreover, we consider 12 Android metrics to capture characteristics exclusive to Android applications.

Source code metrics: We extract those metrics from the source code. We consider different groups of metrics: a) Object-oriented metrics which include the metrics proposed by Chidamber and Kemerer [51] such as Weighted Methods per Class (WMC), Depth Inheritance Tree (DIT), Coupling between objects (CBO), Response for a Class (RFC). b) Readability metrics such as the number of loops and the number of comparisons [37, 240] and c) other source code metrics like the number of Sources Line Of Code (SLOC). To extract these features, we used CK [13] that applies static analysis to calculate code metrics.

Android metrics: We identify 12 metrics exclusive to Android applications. For that, we built a static analysis tool that relies on Spoon [219]. These metrics are:

- *isActivity*: a binary feature that informs whether a class extends the Activity class from the Android API.

[3] We queried the GitHub mirror at 23 of January of 2020.
[4] https://uphf.github.io/FAMAZOA/

- *isView*: a binary feature that informs whether a class extends the View class from the Android API.
- *isBrodcastReceiver*: a binary feature that informs whether a class extends the BroadcastReceiver class from the Android API.
- *isService*: a binary feature that informs whether a class extends the Service class from the Android API.
- *isContentProvider*: a binary feature that informs whether a class extends the ContentProvider class from the Android API.
- *isFragment*: a binary feature that informs whether a class extends the Fragment class from the Android API.
- *isBuildingBlock*: a binary feature that informs whether a class extends one of the essential building blocks (Activity, Service, BrodcastReceiver and ContentProvider) of an Android application.
- *isInAndroidHierarchy*: a binary feature that informs whether a class extends any class from the Android API.
- *Number of parameters coupled*: The number of methods parameters whose type is an object from the Android API.
- *Number of return coupled*: The number of methods whose the return type is an object from the Android API.
- *Number of methods coupled*: The number of methods whose at least one parameter or return type is an object from the Android API.
- *hasAndroidCoupling*: a binary feature that informs whether a class has at least one method coupled.

The total of 54 features listed in Table 6.1 is used to train our model. To extract these features, we created another tool that takes as input a Git repository and the list of commits with migration. This tool relies on jGit, a pure Java library implementing the Git version control system.[5] The tool clones the software repository, then it navigates through all commits. Let $C_r = \{c_1, c_2, ..., c_n\}$ be the set of commits with migrations of a given repository. $\forall c, c \in C$ the tool checkout the source code, then it extracts the 54 features by calling CK [13] and our Android features detector tool. Since the tool operates at the commit information, it also classifies files between migrated or not using the same strategy applied to identify migration used during the creation of the datasets (See 6.1.2.1). When a repository is analyzed, our tool generates a JSON file. This file has for each commit, the values for feature extracted grouped by file.

We successfully extracted features from 1445 projects. Due to exceptions generated by jGit we could not analyze 12 projects (4 projects from $Android_{j2k}$ and 8 from $GitHub_{j2k}$). We ended this process with 1 993 369 files analyzed. A total of 35 929 file migrations was identified. Table 6.2 shows this result grouped by dataset.

[5] https://www.eclipse.org/jgit/

Subcategory	Metric name
Size	Source Lines Of Code (SLOC), Number of methods, Number of fields
Complexity	Weight Method Class (WMC), Max nested blocks
Coupling	Coupling between objects (CBO), Response for a Class (RFC)
Encapsulation	Number of public fields, Number of public methods
Cohesion	Lack of Cohesion of Methods (LCOM), Tight class cohesion (TCC), Loose Class Cohesion (LCC)
Inheritance	Depth Inheritance Tree (DIT)
Readability	Number o unique words, Number of loops, Number of assignments, Number of math operations, Number of comparisons, Number of string literals, Quantity of numbers
Android	isActivity, isView, isBroadcastReceiver, isService, isContentProvider, isFragment, isBuildingBlock, isInAndroidHierarchy, hasAndroidCoupling, Number of methods coupled, Number of parameters coupled, Number of returns coupled
Java-specific metric	Number of default fields, Number of default methods, Number of final fields, Number of final methods, Number of static fields, Number of static methods, Number of private fields, Number of private methods, Number of protected fields, Number of protected methods, Number of abstract methods, Number of anonymous classes, Number of inner classes, Number of lambdas, Number of static invocation (NOSI), Number of synchronized fields, Number of synchronized methods, Number of log statements, Number of parenthesized expressions, Number of returns, Number of try catches, Number of variables

Table 6.1: List of collected features grouped by category.

Table 6.2: Results of the data extraction. For 1% of GitHub projects the tool failed during the extraction.

Dataset	#Project w/ migration	#File migrations
$GitHub_{j2k}$	1 179/1 187 (99%)	27 375/1 495 734 (1%)
$Android_{j2k}$	266/270 (98%)	8 754/497 635 (1%)

6.2 To what extent a classification model learned from migrations done in real projects may recommend migration of files precisely?

Classification models have been applied to several prediction problems of software engineering [85, 8, 18, 74, 15, 44, 242]. Our intuition is that if we are able to learn a classification model from the information of migration done in real projects capable of predicting files that would be migrated, developers could use this model to get recommendations of files to migrate. For that reason, we explored the feasibility of using classification models to recommend file-level migrations.

Section 6.2.1 highlights how classification a model works and describes how we build our models. Section 6.2.2 explains how we evaluated these models, and Section 6.2.3 our evaluation results.

6.2.1 Model training

Predictive analytics is a process in which historical records are used to predict an uncertain future [147]. In the context of machine learning, classification refers to a predictive modeling problem where a class label is predicted for a given example of input data. A classification model attempts to draw some conclusions from observed values. Thus, to build a classification model, it is necessary to feed it with a training dataset that would be used in the learning process. During this process, the model will learn how to best map examples of input to specific class labels. Finally, once the model is built, given one input, it outputs a predicted class label.

In our approach, we feed our model with a dataset that contains real examples of migration. To create this dataset, given a project and its list of commits with migrations, for each commit, we checkout the source code. At this point, we have access to the state of every project file in the state they were on that project version (commit). Then, for every project's file, we extracted the features presented in Section 6.1.3. Consider the example illustrated in Figure 6.5. Project P has n commits. In the first commit (#1), two files (A and B) have been added. Then, after several commits that only affected A and B but did not add or remove any other files, file A was migrated. On that commit, n, beyond the migration of file A, a new file (C) was added. Due to the migration found, we create a row in the training dataset for each project file (A, B, and C). These rows have one attribute for each feature extracted. We recall that to extract the features, we access the project version specified by the commit with migration (n). Moreover, each row has a boolean feature that represents the label that shows whether a file was migrated or not. Therefore, for each file (row in the dataset) given as input, our models predict if that file should be migrated or not based on the file's features. We recall that the classification of a file is based only on the features extracted from that file.

To build our classification models, we consider five different (binary classification) supervised machine learning algorithms, all available in the scikit-learn, a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems [221]: i) Logistic Regression [28], ii) (Gaussian) Naïve Bayes [288], iii) Support Vector Machines [60], iv) Decision Trees [228], and v) Random Forest [29]. The feature set for training these models comprises 54 features: 42 source code metrics and 12 Android metrics. Moreover, we conduct this study using two different datasets: i) $Android_{j2k}$ ii) $GitHub_{j2k} \cup Fama_{j2k}$.

To conduct this study, we execute the following steps:

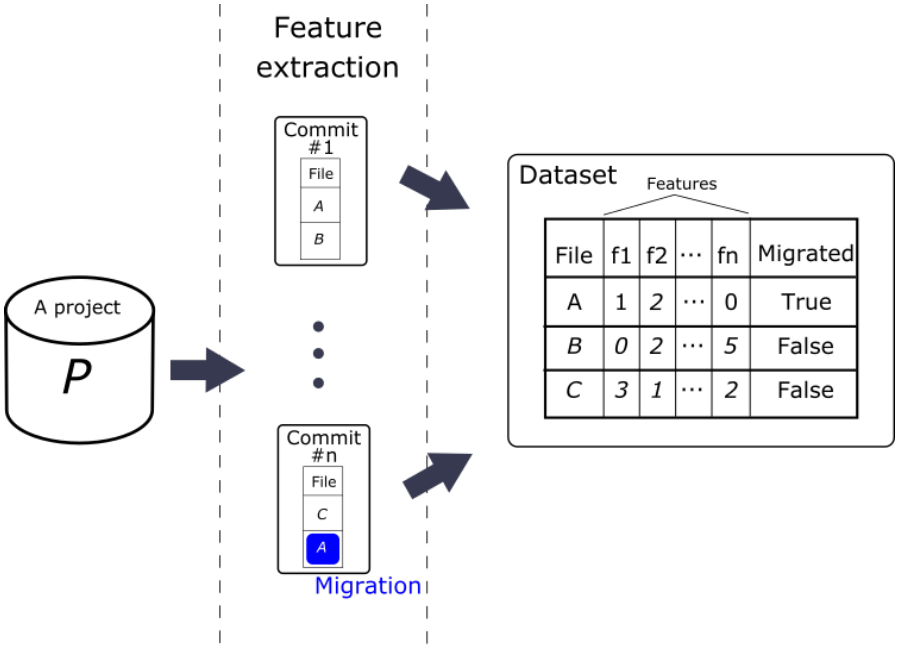


Figure 6.5: The creation of the training dataset for classification. Given P and its list of commits with migration, we checkout the source code for each commit. Then, we extract the features for all project files. Later, this information is formatted. In this format, each file becomes a row in the dataset, and each column contains information related to one feature. There is also one column that indicates if that file was migrated or not.

1. We balance our dataset because some models may under-perform if trained with imbalanced data [111]. As Table 6.2 shows, our dataset is imbalanced. Only 1% of all files are migrated files. For that reason, we use scikit-learn's random under-sampling algorithm, which randomly selects instances of the over-sampled class to perform the balancing.
2. Standardization of datasets is a common requirement for many machine learning estimators implemented in scikit-learn [75]. For that reason, we scale all the features to a [0, 1] range. We use the Min-Max scaler provided by the scikit-learn framework.
3. We use each algorithm's default parameters since our goal is to verify the feasibility of using these algorithms.
4. Finally, we perform a stratified K-fold cross-validation with 10 folds. We split the dataset into ten folds, using 9 folds (90% of the classes) as the training set to build the prediction model and the remaining fold as a validation set to evaluate the accuracy of the model.

6.2.2 Evaluation

Due to the absence of a benchmark of file migrations and the novelty of this work, we could not find a baseline model to compare with our models. For that reason, we used a common practice in machine learning studies, that is, to measure the improvement of a model over random guesses [112]. We used the *DummyClassifier*, a classifier from scikit-learn that makes predictions using simple rules. We set the strategy to generate predictions as 'uniform' to

generate predictions uniformly at random, to serve as a baseline. To answer RQ_9, we reported and compared the mean of three metrics, namely precision, recall and accuracy [20], defined as follows:

$$precision = \frac{TP}{TP + FP} \tag{6.1}$$

$$recall = \frac{TP}{TP + FN} \tag{6.2}$$

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{6.3}$$

where TP (True Positive) is the number of migrated files classified as such by a prediction model; TN (True Negative) denotes the number of files not migrated correctly classified by the model; FP (False Positive) and FN (False Negative) measure the number of files for which a prediction model fails by declaring these files migrated (FP) or not migrated (FN).

6.2.3 Results

In Table 6.3, we show the results of the machine learning algorithms used with their default parameters for each dataset: $Android_{j2k}$ and $GitHub_{j2k}$ combined with $Android_{j2k}$.

First, in this study, we trained our model using $Android_{j2k}$. We observed that when the models were trained only with $Android_{j2k}$, Logistic Regression, SVM Linear and Naïves Baives models had their best performance in terms of precision. Moreover, we concluded all models, Logistic Regression, SVM Linear, Naïves Baives, Decision Trees and Random Forest, are more precise than a random approach with a precision of 0.57, 0.58, 0.60, 0.61 and 0.63, respectively.

In the second part of this study, we trained our models with more information by combining $Android_{j2k}$ and $GitHub_{j2k}$. $Android_{j2k}$ has only Android applications, on the other hand, $GitHub_{j2k}$ has Android applications and other software that migrated files from Java to Kotlin, so when we train a model using $Android_{j2k}$ and $GitHub_{j2k}$ together we are giving more heterogeneous data as input. When we compare the results considering the two training dataset, we note that the dataset's heterogeneity improves the mean recall of every model, as Table 6.3 shows. This heterogeneity improves more the Naïves Bayes, which presented an improvement of 0.69 in terms of recall. However, only Decision Trees and Random Forest take advantage of more data to improve precision and accuracy. Random Forest outperforms Decision Trees considering the three analyzed metrics, precision, recall and accuracy. Therefore, we concluded that with 0.66 of precision, 0.68 of recall, and 0.66 of accuracy, Random Forest is the more suitable model for our approach.

Response to RQ_9**:** *To what extent a classification model learned from migrations done in real projects may recommend migration of files precisely?* The results show that the performance of current machine learning approaches for classifying files that have been migrated outperforms the random approach modestly regardless of the machine learning algorithm. The highest accuracy found was 0.66 for Random Forest when it was trained using the $GitHub_{j2k}$ combined with $Android_{j2k}$ that improves the baseline (random) by 16%.

This empirical study is, to the best of our knowledge, the first one the investigates the performance of classification models in predicting file-level migrations. The results show that compared with a random approach, our models presented a modest improvement. Two models

Table 6.3: The precision (Pr), recall (Re), and accuracy (Acc) of the different machine learning models and their training and test datasets. ↑ indicates that a metric has improved when compared with the empirical study using only $Android_{j2k}$ dataset. Analogously, ↓ indicates that a metric has decreased.

Algorithm	Dataset					
	$Android_{j2k}$			$GitHub_{j2k}$ + $Android_{j2k}$		
	Pr	Re	Acc	Pr	Re	Acc
Baseline (random)	0.48	0.48	0.48	0.50 ↑	0.50 ↑	0.50 ↑
Logistic Regression	0.57	0.46	0.55	0.55 ↓	0.55 ↑	0.55
SVM Linear	0.58	0.41	0.55	0.57 ↓	0.46 ↑	0.55 ↓
Naïves Bayes	0.60	0.14	0.52	0.51 ↓	**0.83** ↑	0.50 ↓
Decision Trees	0.61	0.51	0.59	0.64 ↑	0.55 ↑	0.62 ↑
Random Forest	0.63	0.63	0.63	**0.66** ↑	0.68 ↑	**0.66** ↑

(Decision Trees and Random Forest) presented an improvement greater than 0.1, i.e., comparing with the random approach, they improved in more than 10% their precision, recall, and accuracy. Nevertheless, we consider this result the first step into long-term research towards a model capable of predicting precisely file-level migration and towards a (semi) automated process of migration of applications. This result establishes the initial baseline on the suggestion of file migrations.

6.3 To what extent a learning-to-rank model learned from migrations done in real projects may recommend migration of files precisely?

In this section, we present an empirical study based on another machine learning technique: Learning-To-Rank (LTR). While machine learning classification model that we used in the previous study solves a prediction problem on a single instance at a time, i.e., a classification model looks at all the features associated with a file and classifies it as migrated or not. Learning-to-rank solves a ranking problem on a list of items to come up with an optimal ranking of those items. We explore this technique because we hypothesize that the decision to migrate or not only one file is made considering a project's context and not a file individually, as does the classification technique presented before.

To illustrate our intuition, consider the following example. Suppose that a project composed of 3 files (*A*, *B*, and *C*) needs to be migrated, but the development team cannot perform a *one-step* migration due to resource limitation. Consequently, it is necessary to choose one file to start the migration process. Considering that *A* is the smallest file (SLOC) and *C* the largest, one can decide to migrate *A* by looking individually to that file (as classification performs). However, imagine that *B* is a test file for file *C*. In this scenario, migrate *C* is better than migrate *A* because it is possible to test if the project is still working after the migration. Therefore, our intuition is that using learning-to-rank, our model can capture that knowledge to decide which file migrates first.

To evaluate that approach's feasibility, we conducted an empirical study that applies learning-to-rank algorithms to build a model to recommend file-level migrations. Our goal is to define

a learning-to-rank model learned from source code metrics of real migrations that given an application written in Java (it may have Kotlin code) that should be migrated to Kotlin, it would be capable of rank the applications' files where the files in the top should be the first migrated.

Section 6.3.1 highlights how the learning-to-rank model works and describes how we build our models. Section 6.3.2 explains how we evaluated these models, and Section 6.3.3, presents our evaluation results.

6.3.1 Model training

Given an application that should be migrated to Kotlin, our approach generates a rank of files containing all application files, where the top files should be migrated first. To create such an approach, we need to train a ranking model. In this work, we create such a model by feeding it with information from migrations done by developers. For that, we applied the heuristics presented in 6.1.2.1 to create two datasets that contain real cases of file migrations performed by developers (See Section 6.1.2). Then, we used a learning-to-rank algorithm to learn such a model.

In learning-to-rank, the training data consists of queries and documents where each *query* is associated with a set of *documents*. Moreover, the documents' relevance with respect to the query is also given, in general, represented by a label [159]. In our context, each commit with migration from our datasets becomes a *query*. For each query, its documents are all files of the project in the state they were immediately before the migration, i.e., in the commit before the migration. These documents are labeled according to the actions done on the migration commit (more precisely, if the commit has migrated a file or not).

To illustrate how we transform the information extracted from commit with migration in our training dataset, let us imagine an application with 3 Java files (X.java, Y.java, Z.java) and one Kotlin file (H.kt). We recall that Java and Kotlin are interoperable, so that means that, for example, code from file X.java can invoke functionality from file H as it was written in Java. Consider a commit that performs these actions: i) removes "X.java" ii) updates "Y.java" and iii) adds "X.kt". This commit has, according to heuristic 6.1.2.1, a migration (X was migrated from Java to Kotlin). Consequently, we label these documents as follows: X as migrated, Y, Z as not migrated. From that information, we create a *query*. We repeat this step for each commit with migration, creating one query per each commit. Note that this model captures the relation of the features from the files migrated from those that are not migrated.

We trained our model using the information extracted from the $GitHub_{j2k}$ dataset. This training set contains 7 275 commits with at least one migration identified and a median of 100 files migrated per commit. To create our ranking models, we used XGBoost, a scalable machine learning system for tree boosting proposed by Chen and Guestrin [49]. Specifically, we applied LambdaMART [36] that was developed by Microsoft and it has proven to be very successful in solving real-world ranking problems [36].

6.3.2 Evaluation

To evaluate the performance of our models, we use $Android_{j2k}$ as the testing dataset. Given a test dataset of commits with migrations, testing the model means computing a ranking of the source code files for each commit in the dataset. For any given commit c, the system ranking is created by computing the weighted scoring function $f(c,s)$ for each source code file s, followed by ranking all the files in descending order of their scores. The system ranking is then compared with the ideal ranking in which the relevant files are listed at the top.

Figure 6.6 illustrates how our evaluation works. Given as input a project which in its last version (commit #2) has 4 Java files (A, C, D and E). Since commit #2 migrates one file (A), it becomes a query that contains one document per file in our training dataset. Our approach generates a ranking of files containing all project files on that version (commit #2), i.e., a ranking of documents composed of A, C, D and E. This ranking is compared with the ideal ranking. In this example, the ideal ranking has this order: A (migrated), followed by C, D and E (which are not migrated). Note that since C, D and E are not migrated (documents not relevant), their order does not impact performance measurement.

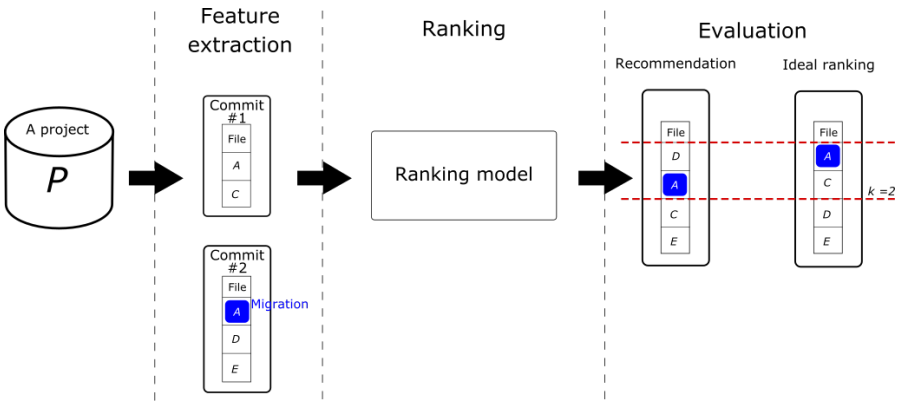


Figure 6.6: For each commit in the training dataset, our approach generates a ranking with all project files. This ranking is compared to an ideal ranking created from the corresponding commit, where the migrated files are in the top. In this example, we set $k = 2$, so only the first two positions of the rankings are considered in the evaluation.

The overall approach performance is then computed using the Mean Average Precision at K ($MAP@k$), i.e., the mean of the Average Precision at K (APK) metric across all instances in the dataset. APK is a measure of the average relevance scores of a set of the top-K documents (file) presented in response to a query (commit). $MAP@k$ ranges from 0 to 1, where a perfect ranking results in MAP equals 1. For each query instance, we compare the set of top-K results with the set of actual relevant documents, that is, a ground truth set of relevant documents for the query. As the median of files migrated per commit is 1, we considered k ranging from 1 to 10. Moreover, we compute our approach's performance improvement by comparing the performance of our approach with the random ranking using the formula: $Improvement = \frac{O-B}{O}$, where O denotes the ranking performance of our approach, B means the ranking performance of a baseline ranking schema.

Due to the absence of a benchmark of file migrations using file ranking, we measure our model's improvement over random guesses.

6.3.3 Results

This section presents the evaluation results of a random approach and LambdaMART applied to rank file-level migrations. Table 6.4 summarizes our results. Our results show that when k increases, MAP increases for both approaches. That makes sense since a greater k means that a model has more chances to select a file correctly in the ranking. For instance, consider a commit with 50 files whose ten files were migrated. When $k = 1$, the model has one chance to put 1 of the ten files migrated in the ranking. When $k = 2$, the model has two chances to put 1 of the ten

files migrated in the ranking.

Table 6.4: Mean Average Precision (MAP) at K of a random approach and LambdaMART.

Algorithm	Mean Average Precison (MAP) at K									
k:	1	2	3	4	5	6	7	8	9	10
Random	0.009	0.030	0.047	0.057	0.061	0.054	0.055	0.053	0.057	0.056
LambdaMART	0.049	0.071	0.083	0.092	0.099	0.105	0.109	0.113	0.117	0.120
Improvement	81%	56%	43%	38%	38%	48%	50%	53.0%	51.0%	53.0%

We also found that our approach outperforms the random approach for any value of k, presenting an improvement of at least 38% for any value of k.

Response to RQ_{10}: *To what extent a learning-to-rank model learned from migrations done in real projects may recommend migration of files precisely?*
The results show that the performance of our learning-to-rank approach to recommend file-level is substantially limited. Its best performance presented a $MAP@10 = 0.11$, but a perfect ranking implies $MAP = 1$.

This empirical study is the first attempt to apply learning-to-rank algorithms to create a recommendation system of file-level migrations to the best of our knowledge. Comparing our model with a random approach, we note that our model outperforms a random approach significantly. However, our results also show that there is room to improve our approach in order to get a better ranking as a result. In Section 6.4, we discuss more in details perspectives to improve our results. Therefore, we consider this result establishes the initial baseline for future research.

6.4 Discussion

This chapter presented a study investigating the feasibility of applying machine learning classification and learning-to-rank to build an approach to recommend file-level migrations of Android applications. Our results showed that both techniques modestly outperform random approaches. Therefore, the results show that there is room to improve these models. Nevertheless, we highlight our approach's novelty and argue that these results establish a baseline for future work. Moreover, it opens directions for researchers.

One strategy to potentially improve our results is to perform a hyperparameter tuning. Each algorithm has a set of parameters, each one having its domain, which may have different types (i.e., continuous, discrete, boolean and nominal), making the entire set of parameters of an algorithm a large space to explore. Consequently, the search for the machine learning algorithm's best parameters is a difficult task in terms of computation complexity, time, and efforts [18]. In future work, we plan to explore different techniques of hyperparameter tuning.

Another aspect researchers may focus on are pre-processing techniques to handle the imbalance of our migration dataset since they can be more important than classifier choice [2]. In our work, we applied the under-sampling strategy to balance our dataset. However, other software engineering studies have used Synthetic Minority Over-sampling TEchnique (SMOTE) to fix the data imbalance [135, 222, 273, 257, 25].

Since our machine learning models achieve a modest performance, as future work, we intend to focus on feature engineering because the current set of features might not be adequate to discriminate between migrated or non-migrated files. Discarding some features or adding new ones could result in a better set of features that may improve our results. Therefore, more research should be conducted to i) evaluate the current set of features and possibly discard some feature, ii) verify to what extent existing metrics applied in other domains of software engineering, like process metrics [281, 118] and ownership metrics [27, 15], are suitable for our problem and iii) develop new metrics able to characterize better migrated or non-migrated file instances.

6.5 Threats to Validity

In this section, we discuss the threats that could affect the validity of our results.

6.5.1 Construct validity

Threats to construct validity concern the relation between theory and observation.

Representativeness of the set of applications collected. To create an accurate machine learning model, a large amount of data is essential. Due to the absence of a benchmark dataset of Java files migration to Kotlin, we have built our dataset. In order to create representative datasets, we mined GitHub, the most popular host of open source projects and the largest public repository of open source Android applications written in Kotlin, FAMAZOA. Since issues might happen during the extraction of a large amount of data, we made both datasets available in our replication package that researchers may use to replicate our study.

Feature selection. The choice of the feature set used to train our machine learning models directly impacts its results, depending on whether these features discriminate adequately, files migrated and non-migrated. However, to the best of our knowledge, there is no study that establishes a relationship between any metric and source code migration. For that reason, we target source code metrics that have been used in a wide variety of experiments like fault prediction, testing, refactoring to assess the overall quality of the software [138, 247, 80, 241, 113, 249, 15, 94]. Moreover, we consider 12 exclusive Android features that, according to our experience with Android development, could support the decision to perform a file migration. Nevertheless, the addition of the new feature may change our results.

Imbalaced data. Due to the fact that there are more instances of the non-migrated files in our dataset than files migrated, we had to face an imbalanced dataset. Given that our datasets are the first datasets of Java projects with file migrations to Kotlin, and some models may under-perform if trained with imbalanced data [111], we decided to perform under-sampling. That is, we chose to remove instances from the over-represented class using random under-sampling. However, we have to admit that other resampling techniques may change our results.

6.5.2 Internal validity

Threats to internal validity concern all the factors that could have impacted our results.

Training pipeline. Our pipeline to train classification models performs scaling and undersampling. To improve our results, we may apply feature reduction, different balancing strategies, and extensive hyperparameter search.

6.5.3 External validity

Threats to external validity concern the generalizability of our findings.

Representativeness of our datasets. Our work relies on two datasets of open source software. However, open source software is a small parcel of the existing software. For instance, FAMOZOA represents only a small parcel of applications published on Google Play. This fact may limit the generalization of our findings. However, we emphasize that we consider the largest repository of open source software, GitHub, and the largest public repository of open source Android applications, to build the dataset used in this work. Nevertheless, both datasets contain only open source applications, which limit the generalization of our findings.

6.6 Summary

In this chapter, we proposed an approach to support developers in the gradual migration of their applications based on machine learning. Our approach aims to assist developers in the gradual migration of their applications, a time-consuming and challenging task [45], by recommending file-level migrations. We investigate its feasibility on recommend file-level migrations of Android applications. Particularly, we analyzed the applicability of our approach using two machine learning techniques: classification and learning-to-rank. For that, we created two datasets of open source projects: i) a dataset of open source projects with migrations from Java to Kotlin, and ii) a dataset of open source Android projects with migrations from Java to Kotlin. Then, for every project's file of both datasets, we extracted 42 source-code metrics and 12 Android metrics.

Using this information as a training dataset, in our first study, we built five classification models: i) Logistic Regression [28], ii) (Gaussian) Naïve Bayes [288], iii) Support Vector Machines [60], iv) Decision Trees [228], and v) Random Forest [29]. Then, we compared them with a random approach in terms of precision, recall and accuracy. The result showed that compared with a random approach, our models presented a modest improvement. Decision Trees and Random Forest presented an improvement greater than 10% in precision, recall, and accuracy compared with the random approach.

In our second study, we reused the training dataset to build a learning-to-rank model. We compared this model with the random approach, and we concluded that it outperforms the random approach. However, we concluded that its performance is substantially limited since its best performance presented a $MAP@10 = 0.11$, but a perfect ranking implies in $MAP = 1$.

These empirical studies are the first attempts to apply learning-to-rank algorithms to create a recommendation system of file-level migrations to the best of our knowledge. Thus, this work allowed us to show that there is room to improve our approach regardless of the machine learning technique.

Since most Android applications are written in Java and the need to move towards Kotlin to keep updated with Android platform news features, we affirm that our approach may significantly impact Android applications' development. Therefore, we consider this result is the first step into long-term research towards a model capable of predicting precisely file-level migration and towards a (semi) automated process of migration of applications. We consider that our results establish the initial baseline on file migrations.

Chapter 7

Conclusion and perspectives

In this chapter, we summarize our thesis's contributions, and we discuss our short- and long-term perspectives.

7.1 Summary of contributions

We presented in this thesis 5 contributions to the research of software engineering applied to development of Android applications using Kotlin.

In Chapter 3, we presented an empirical study that explored the adoption of Kotlin by Android developers. In this study, we created a methodology for detecting applications written in Kotlin. Using this methodology, we analyzed 2 167 applications that we collected from three datasets of Android applications and their 19 838 versions. We found that 11% of the studied Android applications have adopted Kotlin. Moreover, we showed that among the applications that adopted Kotlin, most of them are almost entirely written in Kotlin (80% of its codebase). Additionally, as a result of this study, we created FAMAZOA, a dataset of open source Android applications written in Kotlin.

In Chapter 4, we presented an empirical study that compared the quality of Java-based and Kotlin-based Android applications in terms of the occurrence of code smells. We investigated the presence of ten code smells, 4 Android-specific and 6 object-oriented, using PAPRIKA, a static analysis tool. The results showed that, in general, Java applications have more entities affected by all object-oriented code smells (4) and by one Android code smells. Moreover, we concluded that the introduction of Kotlin code in Android applications initially written in Java produces a rise in the quality scores from, at least, the 50% of the Android applications. More precisely, for 8 and 4 smells (out of 10), the first commits with Kotlin code produce a rise in the quality in, at least, the 62% and 76% of the studied applications, respectively. This fact suggests that the migration of applications to Kotlin may reduce the occurrences of code smells and improve the its quality.

In Chapter 5, we presented two empirical studies focused on the evolution of Kotlin code into Android applications. The result of the first study showed that once Kotlin is introduced into an application, Kotlin tends to become the repository's dominant language. Moreover, we found that 25% of Android applications that introduced Kotlin have entirely migrated. Therefore, the first study shows that the amount of Kotlin code in Android applications tends to increase in the next years, likewise the number of migrated applications. Our second study focused on the usage

evolution of Kotlin features. Its results showed that Android developers are using all Kotlin features that are not available in Java, including the experimental ones. Furthermore, it shows that the number of instances of features tends to grow throughout applications' evolution. These results confirm that the community of Android developers has been using Kotlin consistently.

In Chapter 6, we presented a novel approach to assist the migration of applications from Java to Kotlin. We evaluated this approach's feasibility, applying two different machine learning techniques: classification and learning-to-rank. Our results showed that both techniques outperform a random approach. The best classification model, Random Forest, showed a 16% of improvement in accuracy when compared with a random approach. Our learning-to-rank model outperforms a random approach with at least 50%, but this result is still limited compared to the optimal values. To the best of our knowledge, our study is the first one the investigates the performance of classification and learning-to-rank models on file-level migrations. Since the migration from Java to Kotlin may positively impact the application's maintenance [71] and that activity is a time-consuming and challenging task [45], we consider this result is the first step into long-term research. Therefore, these results establish the initial baseline on file migrations.

7.2 Short-term perspectives

In this section, we present the short-term perspectives of our thesis. We mentioned some perspectives when we discussed our contributions and their implications. The objective of this section is to develop these perspectives and aggregate them.

7.2.1 Kotlin bad practices and code smells

In our study about the usage of Kotlin features (Chapter 5), we observed that developers add more instances along with the evolution of applications. Kotlin brings unique features to the development of Android applications and, consequently, new options for writing applications. However, the misuse of these features could affect the maintainability (quality, performance, etc.) of Android applications. Moreover, as the number of features grows, applications become more prone to overusing features. Therefore, in our future studies, we aim to investigate in-depth the usage of Kotlin features to potentially identify bad practices related to them, especially the most used features. We aim to reproduce the study by Mazinanian et al. [186] to identify the misuse of Kotlin *lambda*. Furthermore, we found instances of *Unsafe call* in most applications, which make them more vulnerable to *NullPointerException*. For that reason, we aim to investigate to what extensions the use of *unsafe call* can be avoided. This type of study can be applied to all Kotlin features. The results of these studies could lead us to identify a catalog of Kotlin-specific or Kotlin-Android-specific code smells.

7.2.2 Power consumption on Kotlin-based Android applications

Similar to the research about Android-specific code smells, the research about the power consumption of Android applications has focused on Java-based applications only [157, 24, 197, 61, 62]. Therefore, we want to understand whether the use of Kotlin impacts this aspect of the development of Android applications.

7.2.3 Feature engineering for assisted migration

The results of the empirical studies presented in Chapter 6 showed that there is room to improve our machine-learned models. A possible way to reach this improvement is to use better features

to feed our models. For that reason, in our future work, we aim to explore feature engineering. Here, we have two possibilities. First, we may explore features that have been used in other software engineering problems like process metrics [281, 118] and ownership metrics [27, 15]. Another possibility is to propose new features. and apply techniques to learn features automatically.

7.3 Long-term perspectives

One of our thesis's main findings is that Android developers are adopting Kotlin, and a relevant portion of applications has been migrated to Kotlin (Chapter 3). To improve our proposed approach to assist developers in migrating applications, we intend to apply feature engineering and find better features to feed our models. Regarding the challenge of migrating applications, our research and short-term perspective covers one aspect of this process, code translation. In the following, we present our perspectives to cover other aspects related to this process.

7.3.1 Test generation for migration

To migrate an application with success, it is necessary to guarantee that the migration process would not modify its behavior. A test suite with good code coverage helps to ensure that any code change would not cause undesired behavior. To complement our approach, as future work, we want to explore test generation techniques such as Randoop [211] and EvoSuite [89] to assist the migration of Android applications.

7.3.2 Automatic software refactoring for migration

Android developers may use the auto converter tool to migrate Java to Kotlin as part of the process of migration. For instance, in the case of Duolingo, to run the IDE's auto converter was the first step of the migration workflow [45]. However, the converted code often needs to be manually modified [181]. Ideally, the source code should be refactored before and after the conversion to simplify its maintenance Terekhov and Verhoef [259]. For that reason, we aim to research automatic software refactoring to verify its applicability in the context of Android applications migration.

7.3.3 Cross-platform development using Kotlin

Kotlin Multiplatform Mobile (KMM) is an SDK for cross-platform mobile development provided by JetBrains [128]. KMM allows developers to use a single codebase for the business logic of iOS and Android apps. Developers only need to write platform-specific code where necessary, for example, to implement a native UI or when working with platform-specific APIs. The shared code, written in Kotlin, is compiled to JVM bytecode with Kotlin/JVM and native binaries with Kotlin/Native, so one can use the KMM business-logic modules just like any other regular mobile library [223]. Moreover, using KMM, developers may migrate their applications code gradually to make them compatible with iOS and Android [128]. Therefore, the use of KMM opens two possibilities related to code migration: i) migration of Java to Kotlin, note that the shared code written in Kotlin is not fully compatible with Java libraries since they are not multi-platform and ii) migration of Object-C/Swift code to Kotlin. As future work, we would like to expand our research to cover these scenarios.

7.4 Final words

This thesis summarizes the research works that we conducted for three years in order to build an understanding of the use of Kotlin to create Android applications. We believe that our work contributes effectively to develop this understanding, and consequently, it helps developers and companies to use Kotlin properly according to their needs. Our work also opens many perspectives for researchers and tool makers. We summarize in 7.1 these perspectives.

Table 7.1: The short and long-term perspectives of our thesis.

#	**Perspectives**
I	Kotlin bad practices and code smells
II	Power consumption on Kotlin-based Android applications
III	Feature engineering for assisted migration
IV	Test generation for migration
V	Automatic software refactoring for migration
VI	Cross-platform development using Kotlin

Bibliography

[1] A.Mahdy AbdelAziz. *Migrating Java enterprise apps to Kotlin*. https://vaadin.com/learn/tutorials/migrate-to-kotlin. (Accessed on 12/11/2020). 2020.

[2] Amritanshu Agrawal and Tim Menzies. "Is "Better Data" Better than "Better Data Miners"? On the Benefits of Tuning SMOTE for Defect Prediction". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 1050–1061. ISBN: 9781450356381. DOI: 10.1145/3180155.3180197.

[3] Jehad Al Dallal. "Object-oriented class maintainability prediction using internal quality attributes". In: *Information and Software Technology* 55 (11) (2013), pp. 2028–2048. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2013.07.005. URL: http://www.sciencedirect.com/science/article/pii/S0950584913001511.

[4] Hamoud Aljamaan, Mahmoud O. Elish, and Irfan Ahmad. "An Ensemble of Computational Intelligence Models for Software Maintenance Effort Prediction". In: *Advances in Computational Intelligence*. Ed. by Ignacio Rojas, Gonzalo Joya, and Joan Gabestany. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 592–603. ISBN: 978-3-642-38679-4.

[5] Kevin Allix, Tegawendé F Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. "Empirical assessment of machine learning-based malware detectors for Android". In: *Empirical Software Engineering* 21 (1) (2016), pp. 183–211. ISSN: 1573-7616. DOI: 10.1007/s10664-014-9352-6.

[6] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "AndroZoo: Collecting Millions of Android Apps for the Research Community Kevin". In: *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16* (2016), pp. 468–471. DOI: 10.1145/2901739.2903508.

[7] Hadeel Alsolai and Marc Roper. "A systematic literature review of machine learning techniques for software maintainability prediction". In: *Information and Software Technology* 119 (September 2019) (2020), p. 106214. ISSN: 09505849. DOI: 10.1016/j.infsof.2019.106214.

[8] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Márcio Ribeiro. "Experience report: Evaluating the effectiveness of decision trees for detecting code smells". In: *2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE 2015* (2016), pp. 261–269. DOI: 10.1109/ISSRE.2015.7381819.

[9] B. Amos, H. Turner, and J. White. "Applying machine learning classifiers to dynamic Android malware detection at scale". In: *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*. 2013, pp. 1666–1671. DOI: 10.1109/IWCMC.2013.6583806.

[10] AndroidDoc. *Array Map | Android Developers*. [Online; accessed 17-July-2018]. 2018. URL: https://developer.android.com/reference/android/support/v4/util/ArrayMap.

[11] AndroidDoc. *Performance Tips | Android Developers*. [Online; accessed 17-July-2018]. 2018. URL: https://developer.android.com/training/articles/perf-tips#PreferStatic.

[12] Peter J. Angeline. "Genetic Programming and Emergent Intelligence". In: *Advances in Genetic Programming*. Cambridge, MA, USA: MIT Press, 1994, pp. 75–97. ISBN: 0262111888.

[13] Maurício Aniche. *Java code metrics calculator (CK)*. Available in https://github.com/mauricioaniche/ck/. 2015.

[14] Maurício Aniche, Gabriele Bavota, Christoph Treude, Arie Van Deursen, and Marco Aurélio Gerosa. "A validated set of smells in model-view-controller architectures". In: *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016* (2017), pp. 233–243. DOI: 10.1109/ICSME.2016.12.

[15] Maurício Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. *The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring*. 2020. arXiv: 2001.03338 [cs.SE].

[16] Apache. *Apache Cordova Official Website*. https://cordova.apache.org/. (Accessed on 11/10/2020).

[17] AppBrain. *Android and Google Play statistics*. https://www.appbrain.com/stats. (Accessed on 11/10/2020).

[18] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. "Comparing and experimenting machine learning techniques for code smell detection". In: *Empirical Software Engineering* 21 (3) (2016), pp. 1143–1191. ISSN: 15737616. DOI: 10.1007/s10664-015-9378-4.

[19] Luca Ardito, Riccardo Coppola, Giovanni Malnati, and Marco Torchiano. "Effectiveness of Kotlin vs. Java in android app development tasks". In: *Information and Software Technology* 127 (2020), p. 106374. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106374. URL: http://www.sciencedirect.com/science/article/pii/S0950584920301439.

[20] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*. Vol. 463. ACM press New York, 1999.

[21] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot". In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*. SOAP '12. Beijing, China: ACM, 2012, pp. 27–38. ISBN: 978-1-4503-1490-9. DOI: 10.1145/2259051.2259056.

[22] Márcio P. Basgalupp, Rodrigo C. Barros, André C. P. L. F. de Carvalho, Alex A. Freitas, and Duncan D. Ruiz. "LEGAL-Tree: A Lexicographic Multi-Objective Genetic Algorithm for Decision Tree Induction". In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC '09. Honolulu, Hawaii: Association for Computing Machinery, 2009, pp. 1085–1090. ISBN: 9781605581668. DOI: 10.1145/1529282.1529521.

[23] Márcio P. Basgalupp, Rodrigo C. Barros, and Duncan D. Ruiz. "Predicting Software Maintenance Effort through Evolutionary-Based Decision Trees". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. Trento, Italy: Association for Computing Machinery, 2012, pp. 1209–1214. ISBN: 9781450308571. DOI: 10.1145/2245276.2231966.

[24] R. J. Behrouz, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann. "EcoDroid: An Approach for Energy-Based Ranking of Android Apps". In: *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. 2015, pp. 8–14. DOI: 10.1109/GREENS.2015.9.

[25] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah. "MAHAKIL: Diversity Based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction". In: *IEEE Transactions on Software Engineering* 44 (6) (2018), pp. 534–550. DOI: 10.1109/TSE.2017.2731766.

[26] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. "Learning-to-Rank vs Ranking-to-Learn : Strategies for Regression Testing in Continuous Integration". In: *International Conference on Software Engineering* (2020).

[27] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. "Don't touch my code! Examining the effects of ownership on software quality". In: *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering* (2011), pp. 4–14. DOI: 10.1145/2025113.2025119.

[28] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[29] Leo Breiman. "Random Forests". In: *Machine Learning* 45 (1) (2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324.

[30] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[31] Andrey Breslav. *Kotlin 1.0 Released: Pragmatic Language for JVM and Android*. https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/. 2016.

[32] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[33] Timofey Bryksin, Victor Petukhov, Ilya Alexin, Stanislav Prikhodko, Alexey Shpilman, Vladimir Kovalenko, and Nikita Povarov. "Using Large-Scale Anomaly Detection on Code to Improve Kotlin Compiler". In: (3) (2020). arXiv: 2004.01618. URL: http://arxiv.org/abs/2004.01618.

[34] Timofey Bryksin, Victor Petukhov, Kirill Smirenko, and Nikita Povarov. "Detecting Anomalies in Kotlin Code". In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ISSTA '18. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 10–12. ISBN: 9781450359399. DOI: 10.1145/3236454.3236457.

[35] Martin D Buhmann. *Radial basis functions: theory and implementations*. Vol. 12. Cambridge university press, 2003.

[36] Christopher JC Burges. "From RankNet to LambdaRank to LambdaMART: An Overview". In: *Microsoft Research Technical Report* 11 (23-581) (2010), p. 81.

[37] Raymond P. L. Buse and Westley R. Weimer. "Learning a Metric for Code Readability". In: *IEEE Trans. Softw. Eng.* 36 (4) (July 2010), pp. 546–558. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.70.

[38] H. Cai, N. Meng, B. Ryder, and D. Yao. "DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling". In: *IEEE Transactions on Information Forensics and Security* 14 (6) (2019), pp. 1455–1470. DOI: 10.1109/TIFS.2018.2879302.

[39] Paolo Calciati and Alessandra Gorla. "How Do Apps Evolve in Their Permission Requests?: A Preliminary Study". In: *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 37–41. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.64.

[40] Paolo Calciati, Konstantin Kuznetsov, Xue Bai, and Alessandra Gorla. "What Did Really Change with the New Release of the App?" In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: ACM, 2018, pp. 142–152. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196449.

[41] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. "A Classifier of Malicious Android Applications". In: *Proceedings of the 2013 International Conference on Availability, Reliability and Security*. ARES '13. USA: IEEE Computer Society, 2013, pp. 607–614. ISBN: 9780769550084. DOI: 10.1109/ARES.2013.80.

[42] Zherui Cao, Yuan Tian, Tien Duy B. Le, and David Lo. "Rule-based specification mining leveraging learning to rank". In: *Automated Software Engineering* 25 (3) (2018), pp. 501–530. ISSN: 15737535. DOI: 10.1007/s10515-018-0231-z.

[43] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. "Investigating the energy impact of Android smells". In: *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering* (2017), pp. 115–126. DOI: 10.1109/SANER.2017.7884614.

[44] Gemma Catolino, Fabio Palomba, Francesca Arcelli Fontana, Andrea De Lucia, Andy Zaidman, and Filomena Ferrucci. "Improving change prediction models with code smell-related information". In: *Empirical Software Engineering* 25 (1) (2020), pp. 49–95. ISSN: 15737616. DOI: 10.1007/s10664-019-09739-0. arXiv: 1905.10889.

[45] Art Chaidarun. *Migrating Duolingo's Android app to 100% Kotlin*. Online; accessed 17-April-2020. Apr. 2020. URL: https://blog.duolingo.com/migrating-duolingos-android-app-to-100-kotlin/.

[46] Art Chaidarun. *Migrating Duolingo's Android app to 100% Kotlin*. Online; accessed 17-April-2020. Apr. 2020. URL: https://blog.duolingo.com/migrating-duolingos-android-app-to-100-kotlin/.

[47] Rivu Chakraborty. *Reactive Programming in Kotlin*. Packt Publishing Ltd, 2017.

[48] Carl Chapman and Kathryn T. Stolee. "Exploring Regular Expression Usage and Context in Python". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: ACM, 2016, pp. 282–293. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931073.

[49] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785.

[50] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. "Detecting performance anti-patterns for applications developed using object-relational mapping". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 1001–1012.

[51] S R Chidamber and C F Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20 (6) (1994), pp. 476–493. ISSN: 0098-5589 VO - 20. DOI: 10.1109/32.295895.

[52] Istehad Chowdhury and Mohammad Zulkernine. "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities". In: *Journal of Systems Architecture* 57 (3) (2011). Special Issue on Security and Dependability Assurance of Software Architectures, pp. 294–313. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2010.06.003. URL: http://www.sciencedirect.com/science/article/pii/S1383762110000615.

[53] Mike Cleron. *Android Developers Blog: Android Announces Support for Kotlin*. https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html. (Accessed on 10/30/2020). May 2017.

[54] Norman Cliff. *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[55] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. "Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues". In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR '15. Florence, Italy: IEEE Press, 2015, pp. 134–145. ISBN: 9780769555942.

[56] Massimo Colosimo, Andrea De Lucia, Giuseppe Scanniello, and Genoveffa Tortora. "Evaluating legacy system migration technologies through empirical studies". In: *Information and Software Technology* 51 (2) (2009), pp. 433–447.

[57] *Comparison to Java Programming Language*. Online; accessed 01-July-2019. Apr. 2016. URL: https://kotlinlang.org/docs/reference/comparison-to-java.html.

[58] Thomas D Cook, Donald Thomas Campbell, and Arles Day. *Quasi-experimentation: Design & analysis issues for field settings*. Vol. 351. Houghton Mifflin Boston, 1979.

[59] Riccardo Coppola, Luca Ardito, and Marco Torchiano. "Characterizing the transition to Kotlin of Android apps: a study on F-Droid, Play Store, and GitHub". In: *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics - WAMA 2019* (2019). DOI: 10.1145/3340496.3342759.

[60] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine Learning* 20 (3) (1995), pp. 273–297. ISSN: 1573-0565. DOI: 10.1007/BF00994018.

[61] L. Cruz, R. Abreu, and J. Rouvignac. "Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring". In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 2017, pp. 205–206. DOI: 10.1109/MOBILESoft.2017.21.

[62] Luis Cruz and Rui Abreu. "Using Automatic Refactoring to Improve Energy Efficiency of Android Apps". In: *In Proceedings of the CIbSE XXI Ibero-American Conference on Software Engineering* (2018). arXiv: 1803.05889. URL: http://arxiv.org/abs/1803.05889.

[63] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. "On the Feasibility of Online Malware Detection with Performance Counters". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013, pp. 559–570. ISBN: 9781450320795. DOI: 10.1145/2485922.2485970.

[64] Android Developers. *Adopt Kotlin for large teams*. Online; accessed 21-September-2020. July 2020. URL: https://developer.android.com/kotlin/adopt-for-large-teams.

[65] Android Developers. *Android 10 for Developers*. https://developer.android.com/about/versions/10/highlights. (Accessed on 10/13/2020). Mar. 2019.

[66] Android Developers. *Android 7.0 for Developers*. https://developer.android.com/about/versions/nougat/android-7.0. (Accessed on 10/13/2020). Aug. 2016.

[67] Android Developers. *Android 8.0 Features and APIs*. https://developer.android.com/about/versions/oreo/android-8.0. (Accessed on 10/13/2020). Mar. 2017.

[68] Android Developers. *Android KTX*. https://developer.android.com/kotlin/ktx. (Accessed on 10/06/2020). 2020.

[69] Android Developers. *Android's Kotlin-first approach*. Online; accessed 21-September-2020. July 2020. URL: https://developer.android.com/kotlin/first.

[70] Android Developers. *AsyncTask API Reference*. [Online; accessed 3-December-2019]. Nov. 2019. URL: https://developer.android.com/reference/android/os/AsyncTask/.

[71] Android Developers. *Google Home reduces #1 cause of crashes by 33%*. https://developer.android.com/stories/apps/google-home. (Accessed on 12/11/2020). Nov. 2020.

[72] Android Developers. *Kotlin coroutines on Android*. https://developer.android.com/kotlin/coroutines. (Accessed on 10/06/2020). Sept. 2020.

[73] Android Developers. *Use common Kotlin patterns with Android*. Online; accessed 01-July-2019. URL: https://developer.android.com/kotlin/common-patterns#delegate.

[74] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. "Detecting code smells using machine learning techniques: Are we there yet?" In: *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings* 2018-March (2018), pp. 612–621. DOI: 10.1109/SANER.2018.8330266.

[75] Scikit-Learn Documentation. *6.3. Preprocessing data*. https://scikit-learn.org/stable/modules/preprocessing.html#standardization-or-mean-removal-and-variance-scaling. (Accessed on 02/08/2020).

[76] Robert Dyer. *A Large-scale Empirical Study of Java Language Feature Usage*. Technical Report. Iowa State University, 2013.

[77] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. "Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 779–790. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568295.

[78] M. O. Elish and K. O. Elish. "Application of TreeNet in Predicting Object-Oriented Software Maintainability: A Comparative Study". In: *2009 13th European Conference on Software Maintenance and Reengineering*. 2009, pp. 69–78. DOI: 10.1109/CSMR.2009.57.

[79] Mahmoud O Elish, Hamoud Aljamaan, and Irfan Ahmad. "Three empirical studies on predicting software maintainability using ensemble methods". In: *Soft Computing* 19 (9) (2015), pp. 2511–2524. ISSN: 1433-7479. DOI: 10.1007/s00500-014-1576-2.

[80] S. Eski and F. Buzluca. "An Empirical Study on Object-Oriented Metrics and Software Evolution in Order to Reduce Testing Costs by Predicting Change-Prone Classes". In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 2011, pp. 566–571. DOI: 10.1109/ICSTW.2011.43.

[81] Facebook. *React Native · A framework for building native apps using React*. https://reactnative.dev/. (Accessed on 11/10/2020).

[82] P. Feng, J. Ma, C. Sun, X. Xu, and Y. Ma. "A Novel Dynamic Android Malware Detection System With Ensemble Learning". In: *IEEE Access* 6 (2018), pp. 30996–31011. DOI: 10.1109/ACCESS.2018.2844349.

[83] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti. "ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications". In: *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 2016, pp. 1–5. DOI: 10.1109/NTMS.2016.7792435.

[84] Matheus Flauzino, Júlio Veríssimo, Ricardo Terra, Elder Cirilo, Vinicius H. S. Durelli, and Rafael S. Durelli. "Are You Still Smelling It?: A Comparative Study Between Java and Kotlin Language". In: *Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse*. SBCARS '18. Sao Carlos, Brazil: ACM, 2018, pp. 23–32. ISBN: 978-1-4503-6554-3. DOI: 10.1145/3267183.3267186.

[85] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V. Mäntylä. "Code smell detection: Towards a machine learning-based approach". In: *IEEE International Conference on Software Maintenance, ICSM* (2013), pp. 396–399. DOI: 10.1109/ICSM.2013.56.

[86] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[87] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[88] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. "LAMBDAFICATOR: From Imperative to Functional Programming Through Automated Refactoring". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 1287–1290. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486986.

[89] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: Association for Computing Machinery, 2011, pp. 416–419. ISBN: 9781450304436. DOI: 10.1145/2025113.2025179.

[90] Jerome H. Friedman. "Stochastic gradient boosting". In: *Computational Statistics & Data Analysis* 38 (4) (2002). Nonlinear Methods and Data Mining, pp. 367–378. ISSN: 0167-9473. DOI: https://doi.org/10.1016/S0167-9473(01)00065-2. URL: http://www.sciencedirect.com/science/article/pii/S0167947301000652.

[91] Jerome H. Friedman and Jacqueline J. Meulman. "Multiple additive regression trees with application in epidemiology". In: *Statistics in Medicine* 22 (9) (2003), pp. 1365–1381. DOI: https://doi.org/10.1002/sim.1501. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sim.1501.

[92] Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. "Refactoring asynchrony in JavaScript". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 353–363.

[93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable software components*. Addison-Wesley Professional, 1995.

[94] J. Gao, L. Li, T. F. Bissyandé, and J. Klein. "On the Evolution of Mobile App Complexity". In: *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 2019, pp. 200–209.

[95] F.-X. Geiger and I. Malavolta. "Datasets of Android Applications: a Literature Review". In: *ArXiv e-prints* (Sept. 2018). arXiv: 1809.10069 [cs.SE].

[96] Franz-Xaver Geiger, Ivano Malavolta, Luca Pascarella, Fabio Palomba, Dario Di Nucci, and Alberto Bacchelli. "A Graph-based Dataset of Commit History of Real-world Android Apps". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: ACM, 2018, pp. 30–33. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196460.

[97] GitHub. *The state of the Octoverse*. https://web.archive.org/web/20190322190823/https://octoverse.github.com/projects. 2013.

[98] Jeffrey van Gogh. *State of Kotlin on Android - YouTube*. https://www.youtube.com/watch?v=AgPj1Q6D--c&feature=youtu.be&t=652. (Accessed on 10/06/2020).

[99] Bruno Góis Mateus and Matias Martinez. "An empirical study on quality of Android applications written in Kotlin language". In: *Empirical Software Engineering* 24 (6) (2019), pp. 3356–3393. DOI: 10.1007/s10664-019-09727-4.

[100] Google. *Flutter - Beautiful native apps in record time*. https://flutter.dev/. (Accessed on 11/10/2020).

[101] Giovanni Grano, Andrea Di Sorbo, Francesco Mercaldo, Corrado A. Visaggio, Gerardo Canfora, and Sebastiano Panichella. "Android apps and user feedback: a dataset for software evolution and quality improvement". In: *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics - WAMA 2017* (2017), pp. 8–11. DOI: 10.1145/3121264.3121266. URL: http://dl.acm.org/citation.cfm?doid=3121264.3121266.

[102] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. "DeepAM: Migrate APIs with multi-modal sequence to sequence learning". In: *IJCAI International Joint Conference on Artificial Intelligence* (2017), pp. 3675–3681. ISSN: 10450823. arXiv: arXiv:1704.07734v1.

[103] Ciaran Gultnieks. *F-Droid - Free and Open Source Android App Repository*. https://www.f-droid.org/. (Accessed on 10/06/2020). 2010.

[104] T. Gyimothy, R. Ferenc, and I. Siket. "Empirical validation of object-oriented metrics on open source software for fault prediction". In: *IEEE Transactions on Software Engineering* 31 (10) (2005), pp. 897–910. DOI: 10.1109/TSE.2005.112.

[105] Roman Haas and Benjamin Hummel. "Learning to Rank Extract Method Refactoring Suggestions for Long Methods". In: *Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies*. Ed. by Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann. Cham: Springer International Publishing, 2017, pp. 45–56. ISBN: 978-3-319-49421-0.

[106] Chet Haase. *Developing for Android II – Google Developers – Medium*. [Online; accessed 17-July-2018]. 2015. URL: https://medium.com/google-developers/developing-for-android-ii-bb9a51f8c8b9.

[107] Chet Haase. *Google I/O 2019: Empowering developers to build the best experiences on Android Play*. Online; accessed 21-April-2020. May 2019. URL: https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html.

[108] Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. "Code Smells in iOS Apps: How Do They Compare to Android?" In: *Proceedings - 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft 2017* (2017), pp. 110–121. DOI: 10.1109/MOBILESoft.2017.11.

[109] Sarra Habchi, Naouel Moha, and Romain Rouvoy. "The Rise of Android Code Smells: Who is to Blame?" In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 445–456. DOI: 10.1109/MSR.2019.00071.

[110] Sarra Habchi, Romain Rouvoy, and Naouel Moha. "On the Survival of Android Code Smells in the Wild". In: *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems*. MOBILESoft '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 87–98.

[111] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering". In: *IEEE Transactions on Software Engineering* 38 (6) (2012), pp. 1276–1304. DOI: 10.1109/TSE.2011.103.

[112] Peter Harrington. *Machine Learning in Action*. USA: Manning Publications Co., 2012. ISBN: 1617290181.

[113] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. "An Empirical Study on Software Defect Prediction with a Simplified Metric Set". In: *Inf. Softw. Technol.* 59 (C) (Mar. 2015), pp. 170–190. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2014.11.006.

[114] Geoffrey Hecht. "Detection and analysis of impact of code smells in mobile applications". PhD Thesis. Université Lille 1 : Sciences et Technologies ; Université du Québec à Montréal, Nov. 2016. URL: https://tel.archives-ouvertes.fr/tel-01418158.

[115] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. "Tracking the Software Quality of Android Applications Along Their Evolution". In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 236–247. ISBN: 978-1-5090-0025-8. DOI: 10.1109/ASE.2015.46.

[116] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. "An empirical study of the performance impacts of Android code smells". In: *Proceedings of the International Workshop on Mobile Software Engineering and Systems - MOBILESoft '16* (2016), pp. 59–69. ISSN: 9781450321389. DOI: 10.1145/2897073.2897100. arXiv: arXiv:1508.06655v1.

[117] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. "Detecting Antipatterns in Android Apps". In: *Proceedings - 2nd ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft 2015* (2015), pp. 148–149. DOI: 10.1109/MobileSoft.2015.38.

[118] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi. "DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. May 2019, pp. 34–45. DOI: 10.1109/MSR.2019.00016.

[119] Felipe Hoffa. *GitHub on BigQuery: Analyze all the open source code*. Online; accessed 17-April-2020. June 2016. URL: https://cloud.google.com/blog/products/gcp/github-on-bigquery-analyze-all-the-open-source-code.

[120] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*. Vol. 398. John Wiley & Sons, 2013.

[121] David Hovemeyer and William Pugh. "Finding Bugs is Easy". In: *SIGPLAN Not.* 39 (12) (Dec. 2004), pp. 92–106. ISSN: 0362-1340. DOI: 10.1145/1052883.1052895.

[122] Shahid Hussain, Jacky Keung, Mohammad Khalid Sohail, Arif Ali Khan, Manzoor Ilahi, Ghufran Ahmad, Muhammad Rafiq Mufti, and Muhammad Asim Noor. "A methodology to rank the design patterns on the base of text relevancy". In: *Soft Computing* 23 (24) (2019), pp. 13433–13448. DOI: 10.1007/s00500-019-03882-y.

[123] *Inline Classes*. Online; accessed 01-July-2019. Oct. 2018. URL: https://kotlinlang.org/docs/reference/inline-classes.html.

[124] *Inline Functions*. Online; accessed 01-July-2019. Apr. 2016. URL: https://kotlinlang.org/docs/reference/inline-functions.html.

[125] Mike James. *Programmer's Guide To Kotlin*. I/O Press, 2017.

[126] H. A. Al-Jamimi and L. Ghouti. "Efficient prediction of software fault proneness modules using support vector machines and probabilistic neural networks". In: *2011 Malaysian Conference in Software Engineering*. 2011, pp. 251–256. DOI: 10.1109/MySEC.2011.6140679.

[127] JetBrains. *Kotlin Coding Conventions*. https://kotlinlang.org/docs/reference/coding-conventions.html. 2019.

[128] JetBrains. *Kotlin for Cross-Platform Mobile Development*. https://kotlinlang.org/lp/mobile/. (Accessed on 12/02/2020). Dec. 2020.

[129] JetBrains. *Kotlin Language Documentation*. https://kotlinlang.org/docs/kotlin-docs.pdf. 2019.

[130] Tian Jiang, Lin Tan, and Sunghun Kim. "Personalized Defect Prediction". In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE'13. Silicon Valley, CA, USA: IEEE Press, 2013, pp. 279–289. ISBN: 9781479902156. DOI: 10.1109/ASE.2013.6693087.

[131] George H. John and Pat Langley. "Estimating Continuous Distributions in Bayesian Classifiers". In: *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*. UAI'95. Montréal, Qué, Canada: Morgan Kaufmann Publishers Inc., 1995, pp. 338–345. ISBN: 1558603859.

[132] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 21-April-2020]. 2001–. URL: http://www.scipy.org/.

[133] M. Jorgensen and M. Shepperd. "A Systematic Review of Software Development Cost Estimation Studies". In: *IEEE Transactions on Software Engineering* 33 (1) (2007), pp. 33–53. DOI: 10.1109/TSE.2007.256943.

[134] JRebel. *JVM Languages Report: Extended Interview With Kotlin Creator Andrey Breslav*. https://www.jrebel.com/blog/interview-with-kotlin-creator-andrey-breslav. 2013.

[135] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto. "The Effects of Over and Under Sampling on Fault-prone Module Detection". In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. 2007, pp. 196–204. DOI: 10.1109/ESEM.2007.28.

[136] A. Kaur, K. Kaur, and S. Jain. "Predicting software change-proneness with code smells and class imbalance learning". In: *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 2016, pp. 746–754. DOI: 10.1109/ICACCI.2016.7732136.

[137] A. Kaur, K. Kaur, and K. Pathak. "Software maintainability prediction by data mining of software code metrics". In: *2014 International Conference on Data Mining and Intelligent Computing (ICDMIC)*. 2014, pp. 1–6. DOI: 10.1109/ICDMIC.2014.6954262.

[138] Inderpreet Kaur and Arvinder Kaur. "Empirical Study of Software Quality Estimation". In: *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*. CCSEIT '12. Coimbatore UNK, India: Association for Computing Machinery, 2012, pp. 694–700. ISBN: 9781450313100. DOI: 10.1145/2393216.2393332.

[139] Hannes Kegel and Friedrich Steimann. "Systematically Refactoring Inheritance to Delegation in Java". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 431–440. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368147.

[140] Marouane Kessentini and Ali Ouni. "Detecting Android Smells Using Multi-Objective Genetic Programming". In: *Proceedings - 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft 2017* (2017), pp. 122–132. DOI: 10.1109/MOBILESoft.2017.29.

[141] Hammad Khalid, Meiyappan Nagappan, and Ahmed E. Hassan. "Examining the relationship between FindBugs warnings and app ratings". In: *IEEE Software* 33 (4) (2016), pp. 34–39. ISSN: 07407459. DOI: 10.1109/MS.2015.29.

[142] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. "Classification-tree models of software-quality over multiple releases". In: *IEEE Transactions on Reliability* 49 (1) (2000), pp. 4–11. DOI: 10.1109/24.855532.

[143] Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. "Precise Learn-to-Rank Fault Localization Using Dynamic and Static Features of Target Programs". In: *ACM Transactions on Software Engineering and Methodology* 28 (4) (2019), pp. 1–34. ISSN: 1049331X. DOI: 10.1145/3345628.

[144] P. S. Kochhar, D. Wijedasa, and D. Lo. "A Large Scale Study of Multiple Programming Languages and Code Quality". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 563–573.

[145] C. van Koten and A. R. Gray. "An Application of Bayesian Network for Predicting Object-Oriented Software Maintainability". In: *Inf. Softw. Technol.* 48 (1) (Jan. 2006), pp. 59–67. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2005.03.002.

[146] KotlinDoc. *Classes and Inheritance - Kotlin Programming Language*. [Online; accessed 17-July-2018]. 2018. URL: https://kotlinlang.org/docs/reference/classes.html#companion-objects.

[147] Vijay Kotu and Bala Deshpande. *Predictive Analytics and Data Mining: Concepts and Practice with RapidMiner*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN: 0128014601.

[148] Jochen Kreimer. "Adaptive Detection of Design Flaws". In: *Electronic Notes in Theoretical Computer Science* 141 (4) (2005). Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005), pp. 117–136. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2005.02.059. URL: http://www.sciencedirect.com/science/article/pii/S1571066105051844.

[149] Paull Krill. *JetBrains readies JVM-based language*. https://www.infoworld.com/article/2622405/jetbrains-readies-jvm-based-language.html. 2011.

[150] Frederic Lardinois. *Kotlin is now Google's preferred language for Android app development | TechCrunch*. https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development. (Accessed on 12/07/2020). May 2019.

[151] Tien Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. "A learning-to-rank based fault localization approach using likely invariants". In: *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis* (2016), pp. 177–188. DOI: 10.1145/2931037.2931049.

[152] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. "Metrics and Laws of Software Evolution - The Nineties View". In: *Proceedings of the 4th International Symposium on Software Metrics*. METRICS '97. USA: IEEE Computer Society, 1997, p. 20. ISBN: 0818680938.

[153] Meir M Lehman. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68 (9) (1980), pp. 1060–1076.

[154] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings". In: *IEEE Transactions on Software Engineering* 34 (4) (2008), pp. 485–496. DOI: 10.1109/TSE.2008.35.

[155] Ian Ni-Lewis. *Avoiding Allocations in onDraw() (100 Days of Google Dev)*. [Online; accessed 17-July-2018]. 2015. URL: https://www.youtube.com/watch?v=HAK5acHQ53E.

[156] Ian Ni-Lewis. *Custom Views and Performance (100 Days of Google Dev)*. [Online; accessed 17-July-2018]. 2015. URL: https://youtu.be/zK2i7ivzK7M?t=4m57s.

[157] D. Li, S. Hao, J. Gui, and W. G. J. Halfond. "An Empirical Study of the Energy Consumption of Android Applications". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 121–130. DOI: 10.1109/ICSME.2014.34.

[158] Deguang Li, Bing Guo, Yan Shen, Junke Li, and Yanhui Huang. "The evolution of open-source mobile applications: An empirical study". In: *Journal of Software: Evolution and Process* 29 (7) (2017), pp. 1–18. ISSN: 20477481. DOI: 10.1002/smr.1855.

[159] Hang Li. "A short introduction to learning to rank". In: *IEICE Transactions on Information and Systems* 94 (10) (2011), pp. 1854–1862.

[160] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. "Static analysis of android apps: A systematic literature review". In: *Information and Software Technology* 88 (2017), pp. 67–95. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.04.001. URL: http://www.sciencedirect.com/science/article/pii/S0950584917302987.

[161] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F. Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. "AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community". In: *ArXiv e-prints* (Sept. 2017). arXiv: 1709.05281 [cs.SE].

[162] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. "Characteristics of Application Software Maintenance". In: *Commun. ACM* 21 (6) (June 1978), pp. 466–471. ISSN: 0001-0782. DOI: 10.1145/359511.359522.

[163] Mario Linares-Vásquez, Sam Klock, Collin McMillan, Aminata Sabané, Denys Poshyvanyk, and Yann-Gaël Guéhéneuc. "Domain Matters: Bringing Further Evidence of the Relationships among Anti-Patterns, Application Domains, and Quality-Related Metrics in Java Mobile Apps". In: *Proceedings of the 22nd International Conference on Program Comprehension*. ICPC 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 232–243. ISBN: 9781450328791. DOI: 10.1145/2597008.2597144.

[164] Alex Lockwood. *How to Leak a Context: Handlers & Inner Classes*. [Online; accessed 17-July-2018]. 2013. URL: https://www.androiddesignpatterns.com/2013/01/inner-class-handler-memory-leak.html.

[165] Pablo Loyola, Kugamoorthy Gajananan, and Fumiko Satoh. "Bug localization by learning to rank and represent bug inducing changes". In: *International Conference on Information and Knowledge Management, Proceedings* (2018), pp. 657–665. DOI: 10.1145/3269206.3271811.

[166] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. "Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations". In: *Universitas Psychologica* 10 (May 2011), pp. 545–555. ISSN: 1657-9267. URL: http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S1657-92672011000200018%5C&nrm=iso.

[167] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, and E. Aimeur. "SMURF: A SVM-based Incremental Anti-pattern Detection Approach". In: *2012 19th Working Conference on Reverse Engineering*. 2012, pp. 466–475. DOI: 10.1109/WCRE.2012.56.

[168] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, G. Antoniol, and E. Aïmeur. "Support vector machines for anti-pattern detection". In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012, pp. 278–281. DOI: 10.1145/2351676.2351723.

[169] I. Malavolta, R. Verdecchia, B. Filipovic, M. Bruntink, and P. Lago. "How Maintainability Issues of Android Apps Evolve". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sept. 2018, pp. 334–344. DOI: 10.1109/ICSME.2018.00042.

[170] Ruchika Malhotra, Arvinder Kaur, and Yogesh Singh. "Empirical validation of object-oriented metrics for predicting fault proneness at different severity levels using support vector machines". In: *International Journal of System Assurance Engineering and Management* 1 (3) (2010), pp. 269–281. ISSN: 0976-4348. DOI: 10.1007/s13198-011-0048-7.

[171] B. A. Malloy and J. F. Power. "Quantifying the Transition from Python 2 to 3: An Empirical Study of Python Applications". In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Nov. 2017, pp. 314–323. DOI: 10.1109/ESEM.2017.45.

[172] Brian A. Malloy and James F. Power. "An empirical analysis of the transition from Python 2 to Python 3". In: *Empirical Software Engineering* 24 (2) (Apr. 2019), pp. 751–778. ISSN: 1573-7616. DOI: 10.1007/s10664-018-9637-2.

[173] N. Maneerat and P. Muenchaisri. "Bad-smell prediction from software design model using machine learning techniques". In: *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*. 2011, pp. 331–336. DOI: 10.1109/JCSSE.2011.5930143.

[174] Umme Ayda Mannan, Iftekhar Ahmed, Rana Abdullah M. Almurshed, Danny Dig, and Carlos Jensen. "Understanding code smells in Android applications". In: *International Workshop on Mobile Software Engineering and Systems (MOBILESoft '16)* (2016), pp. 225–234. DOI: 10.1145/2897073.2897094.

[175] Alessandro Marchetto and Filippo Ricca. "From objects to services: Toward a stepwise migration approach for java applications". In: *International Journal on Software Tools for Technology Transfer* 11 (6) (2009), pp. 427–440. ISSN: 14332779. DOI: 10.1007/s10009-009-0123-4.

[176] Gabriele Mariotti. *AntiPattern: freezing a UI with Broadcast Receiver*. [Online; accessed 17-July-2018]. 2013. URL: http://gmariotti.blogspot.com/2013/02/antipattern-freezing-ui-with-broadcast.html.

[177] Gabriele Mariotti. *Antipattern: freezing the UI with a Service and an IntentService*. [Online; accessed 17-July-2018]. 2013. URL: http://gmariotti.blogspot.com/2013/03/antipattern-freezing-ui-with-service.html.

[178] Gabriele Mariotti. *Antipattern: freezing the UI with an AsyncTask*. [Online; accessed 17-July-2018]. 2013. URL: http://gmariotti.blogspot.com/2013/02/antipattern-freezing-ui-with-asynctask.html.

[179] J. Martin and H. A. Müller. "C to Java migration experiences". In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* (2002), pp. 143–153. ISSN: 15345351. DOI: 10.1109/CSMR.2002.995799.

[180] Matias Martinez and Sylvain Lecomte. "Towards the Quality Improvement of Cross-platform Mobile Applications". In: *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. MOBILESoft '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 184–188. ISBN: 978-1-5386-2669-6. DOI: 10.1109/MOBILESoft.2017.30.

[181] Matias Martinez and Bruno Gois Mateus. *How and Why did developers migrate Android Applications from Java to Kotlin? A study based on code analysis and interviews with developers*. 2020. arXiv: 2003.12730 [cs.SE].

[182] Matias Martinez and Martin Monperrus. "Coming: A Tool for Mining Change Pattern Instances from Git Commits". In: *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 79–82. DOI: 10.1109/ICSE-Companion.2019.00043.

[183] Bruno Gois Mateus and Matias Martinez. *kotlinandroid/multiple_manifest_problem.md at master · UPHF/kotlinandroid · GitHub*. https://github.com/UPHF/kotlinandroid/blob/master/docs/multiple_manifest_problem.md. (Accessed on 10/12/2020). Mar. 2019.

[184] Bruno Gois Mateus and Matias Martinez. "On the Adoption, Usage and Evolution of Kotlin Features in Android Development". In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM '20. Bari, Italy: Association for Computing Machinery, 2020. ISBN: 9781450375801. DOI: 10.1145/3382494.3410676.

[185] Bruno Góis Mateus and Matias Martinez. *FAMAZOA - F-droid AndroidtimeMachine AndroZoo Open-source Applications*. https://uphf.github.io/FAMAZOA/. (Accessed on 10/22/2020). 2018.

[186] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. "Understanding the Use of Lambda Expressions in Java". In: *Proc. ACM Program. Lang.* 1 (OOPSLA) (Oct. 2017), 85:1–85:31. ISSN: 2475-1421. DOI: 10.1145/3133909.

[187] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. "Discovering refactoring opportunities in cascading style sheets". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 496–506.

[188] Colt McAnlis. *Android Performance Patterns: Overdraw, Cliprect, QuickReject*. [Online; accessed 17-July-2018]. 2015. URL: https://www.youtube.com/watch?v=vkTn3Ule4Ps.

[189] Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* (4) (1976), pp. 308–320.

[190] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. "Challenges in software evolution". In: *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. 2005, pp. 13–22. DOI: 10.1109/IWPSE.2005.7.

[191] Bertrand Meyer. *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs, 1997.

[192] Microsfot. *Xamarin | Open-source mobile app platform for .NET*. https://dotnet.microsoft.com/apps/xamarin. (Accessed on 11/10/2020).

[193] Roberto Minelli and Michele Lanza. "Software analytics for mobile applications - Insights & lessons learned". In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* (2013), pp. 144–153. ISSN: 15345351. DOI: 10.1109/CSMR.2013.24.

[194] Ghassan Misherghi and Zhendong Su. "HDD: Hierarchical Delta Debugging". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: Association for Computing Machinery, 2006, pp. 142–151. ISBN: 1595933751. DOI: 10.1145/1134285.1134307.

[195] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells". In: *IEEE Trans. Softw. Eng.* 36 (1) (Jan. 2010), pp. 20–36. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.50.

[196] Rodrigo Morales, Ruben Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. "Anti-patterns and the energy efficiency of Android applications". In: *ArXiv e-prints* (Oct. 2016). arXiv: 1610.05711 [cs.SE].

[197] Rodrigo Morales, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. "EARMO: An Energy-Aware Refactoring Approach for Mobile Apps". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, p. 59. ISBN: 9781450356381. DOI: 10.1145/3180155.3182524.

[198] Marcin Moskala and Igor Wojda. *Android Development with Kotlin*. Packt Publishing Ltd, 2017.

[199] M. Mossienko. "Automated Cobol to Java recycling". In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* 7 (812) (2003), pp. 40–50. ISSN: 15345351. DOI: 10.1109/CSMR.2003.1192409.

[200] Meiyappan Nagappan and Emad Shihab. "Future Trends in Software Engineering Research for Mobile Apps". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2016), pp. 21–32. ISSN: 9781509018550. DOI: 10.1109/SANER.2016.88. URL: http://ieeexplore.ieee.org/document/7476770/.

[201] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. "Statistical learning approach for mining API usage mappings for code migration". In: *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (2014), pp. 457–468. DOI: 10.1145/2642937.2643010.

[202] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. "Migrating code with statistical machine translation". In: *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings* (2014), pp. 544–547. DOI: 10.1145/2591062.2591072.

[203] Haoran Niu, Iman Keivanloo, and Ying Zou. "Learning to rank code examples for code search engines". In: *Empirical Software Engineering* 22 (1) (2017), pp. 259–291. ISSN: 15737616. DOI: 10.1007/s10664-015-9421-5.

[204] JB Nizet. *The story of a Java to Kotlin migration | Ninja Squad*. https://blog.ninja-squad.com/2018/05/22/kotlin-migration/. (Accessed on 12/11/2020). May 2018.

[205] V. Oliveira, L. Teixeira, and F. Ebert. "On the Adoption of Kotlin on Android Development: A Triangulation Study". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 206–216.

[206] Oracle. *Java Platform, Micro Edition (Java ME)*. https://www.oracle.com/java/technologies/javameoverview.html. (Accessed on 09/23/2020). 2020.

[207] H. Osman, A. Chis, C. Corrodi, M. Ghafari, and O. Nierstrasz. "Exception Evolution in Long-Lived Java Systems". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. May 2017, pp. 302–311. DOI: 10.1109/MSR.2017.21.

[208] H. Osman, A. Chiş, J. Schaerer, M. Ghafari, and O. Nierstrasz. "On the evolution of exception usage in Java projects". In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Feb. 2017, pp. 422–426. DOI: 10.1109/SANER.2017.7884646.

[209] Jeffrey L. Overbey and Ralph E. Johnson. "Regrowing a Language: Refactoring Tools Allow Programming Languages to Evolve". In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: Association for Computing Machinery, 2009, pp. 493–502. ISBN: 9781605587660. DOI: 10.1145/1640089.1640127.

[210] Stack Overflow. *Stack Overflow Developer Survey 2020*. Online; accessed 21-September-2020. Apr. 2020. URL: https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted.

[211] Carlos Pacheco and Michael D. Ernst. "Randoop: Feedback-Directed Random Testing for Java". In: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*. OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pp. 815–816. ISBN: 9781595938657. DOI: 10.1145/1297846.1297902.

[212] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. "Mining version histories for detecting code smells". In: *IEEE Transactions on Software Engineering* 41 (5) (2015), pp. 462–489. ISSN: 00985589. DOI: 10.1109/TSE.2014.2372760.

[213] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation". In: *Empirical Software Engineering* 23 (3) (2018), pp. 1188–1221. ISSN: 15737616. DOI: 10.1007/s10664-017-9535-z.

[214] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. "Lightweight detection of Android-specific code smells: The aDoctor project". In: *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering* (2017), pp. 487–491. DOI: 10.1109/SANER.2017.7884659. URL: https://dibt.unimol.it/staff/fpalomba/documents/C18.pdf.

[215] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. "On the impact of code smells on the energy consumption of mobile applications". In: *Information and Software Technology* 105 (2019), pp. 43–55. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.08.004. URL: http://www.sciencedirect.com/science/article/pii/S0950584918301678.

[216] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. "Adoption and use of Java generics". In: *Empirical Software Engineering* 18 (6) (Dec. 2013), pp. 1047–1089. ISSN: 1573-7616. DOI: 10.1007/s10664-012-9236-6.

[217] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. "Java Generics Adoption: How New Features are Introduced, Championed, or Ignored". In: *Proceedings of the International Working Conference on Mining Software Repositories*. ACM, May 2011. URL: https://www.microsoft.com/en-us/research/publication/java-generics-adoption-how-new-features-are-introduced-championed-or-ignored/.

[218] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. "SPOON: A Library for Implementing Analyses and Transformations of Java Source Code". In: *Softw. Pract. Exper.* 46 (9) (Sept. 2016), pp. 1155–1179. ISSN: 0038-0644. DOI: 10.1002/spe.2346.

[219] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code". In: *Software: Practice and Experience* (2015), na. DOI: 10.1002/spe.2346.

[220] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. "Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection". In: *Proceedings of the 27th International Conference on Program Comprehension*. ICPC '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 93–104. DOI: 10.1109/ICPC.2019.00023.

[221] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *CoRR* abs/1201.0490 (2012). arXiv: 1201.0490. URL: http://arxiv.org/abs/1201.0490.

[222] L. Pelayo and S. Dick. "Evaluating Stratification Alternatives to Improve Software Defect Prediction". In: *IEEE Transactions on Reliability* 61 (2) (2012), pp. 516–525. DOI: 10.1109/TR.2012.2183912.

[223] Ekaterina Petrova. *Kotlin Multiplatform Mobile Goes Alpha – Kotlin Blog*. https://blog.jetbrains.com/kotlin/2020/08/kotlin-multiplatform-mobile-goes-alpha/. (Accessed on 12/02/2020). Aug. 2020.

[224] G. Pinto, K. Liu, F. Castor, and Y. D. Liu. "A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Oct. 2016, pp. 20–31. DOI: 10.1109/ICSME.2016.34.

[225] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. "A Large-scale Study on the Usage of Java's Concurrent Programming Constructs". In: *J. Syst. Softw.* 106 (C) (Aug. 2015), pp. 59–81. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.04.064.

[226] Peter Pirkelbauer, Damian Dechev, and Bjarne Stroustrup. "Source Code Rejuvenation Is Not Refactoring". In: *SOFSEM 2010: Theory and Practice of Computer Science*. Ed. by Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 639–650. ISBN: 978-3-642-11266-9.

[227] *Press Duolingo*. Online; accessed 29-May-2020. Apr. 2020. URL: https://www.duolingo.com/press.

[228] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[229] M. El-Ramly, R. Eltayeb, and H. A. Alla. "An Experiment in Automatic Conversion of Legacy Java Programs to C#". In: *Proceedings of the IEEE International Conference on Computer Systems and Applications*. AICCSA '06. USA: IEEE Computer Society, 2006, pp. 1037–1045. ISBN: 1424402115. DOI: 10.1109/AICCSA.2006.205215.

[230] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. "A Large Scale Study of Programming Languages and Code Quality in Github". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 155–165. ISBN: 9781450330565. DOI: 10.1145/2635868.2635922.

[231] M. Rebouças, G. Pinto, F. Ebert, W. Torres, A. Serebrenik, and F. Castor. "An Empirical Study on the Usage of the Swift Programming Language". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. Mar. 2016, pp. 634–638. DOI: 10.1109/SANER.2016.66.

[232] Jan Reimann, Martin Brylski, and Uwe Aßmann. "A tool-supported quality smell catalogue for android developers". In: *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*. Vol. 2014. 2014.

[233] Jan Reimann, Martin Brylski, and Uwe Aßmann. *Android smells catalogue*. [Online; accessed 17-July-2018]. 2014. URL: http://www.modelrefactoring.org/smell_catalog.

[234] M. P. Robillard and K. Kutschera. "Lessons Learned While Migrating From Swing to JavaFX". In: *IEEE Software* 37 (3) (2020), pp. 78–85.

[235] D. Romano and M. Pinzger. "Using source code metrics to predict change-prone Java interfaces". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 303–312. DOI: 10.1109/ICSM.2011.6080797.

[236] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen'sd indices the most appropriate choices". In: *annual meeting of the Southern Association for Institutional Research*. Citeseer. 2006.

[237] Rubén Saborido, Rodrigo Morales, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "Getting the most from map data structures in Android". In: *Empirical Software Engineering* 23 (5) (2018), pp. 2829–2864. DOI: 10.1007/s10664-018-9607-8. URL: https://doi.org/10.1007/s10664-018-9607-8.

[238] J. Sahs and L. Khan. "A Machine Learning Approach to Android Malware Detection". In: *2012 European Intelligence and Security Informatics Conference*. 2012, pp. 141–147. DOI: 10.1109/EISIC.2012.34.

[239] Stephen Samuel and Stefan Bocutiu. *Programming kotlin*. Packt Publishing Ltd, 2017.

[240] Simone Scalabrino, Mario Linares-Vasquez, Rocco Oliveto, and Denys Poshyvanyk. "A Comprehensive Model for Code Readability". In: *Journal of Software: Evolution and Process* 26 (12) (2017), pp. 1–29. ISSN: 20477481. DOI: 10.1002/smr. arXiv: 1408.1293.

[241] R. Shatnawi. "Empirical study of fault prediction for open-source systems using the Chidamber and Kemerer metrics". In: *IET Software* 8 (3) (2014), pp. 113–119. DOI: 10.1049/iet-sen.2013.0008.

[242] Ying Shi, Ming Li, Steven Arndt, and Carol Smidts. "Metric-based software reliability prediction approach and its application". In: *Empirical Software Engineering* 22 (4) (2017), pp. 1579–1633. ISSN: 15737616. DOI: 10.1007/s10664-016-9425-9.

[243] Zhendong Shi, Jacky Keung, Kwabena Ebo Bennin, and Xingjun Zhang. "Comparing learning to rank techniques in hybrid bug localization". In: *Applied Soft Computing Journal* 62 (2018), pp. 636–648. ISSN: 15684946. DOI: 10.1016/j.asoc.2017.10.048.

[244] Yonghee Shin and Laurie Williams. "An Initial Study on the Use of Execution Complexity Metrics as Indicators of Software Vulnerabilities". In: *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*. SESS '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 1–7. ISBN: 9781450305815. DOI: 10.1145/1988630.1988632.

[245] Yonghee Shin and Laurie Williams. "Can traditional fault prediction models be used for vulnerability prediction?" In: *Empirical Software Engineering* 18 (1) (2013), pp. 25–59. ISSN: 1573-7616. DOI: 10.1007/s10664-011-9190-8.

[246] Leonardo Humberto Silva, Miguel Ramos, Marco Tulio Valente, Alexandre Bergel, and Nicolas Anquetil. "Does JavaScript software embrace classes?" In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE. 2015, pp. 73–82.

[247] Gagandeep Singh. "Metrics for Measuring the Quality of Object-Oriented Software". In: *SIGSOFT Softw. Eng. Notes* 38 (5) (Aug. 2013), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/2507288.2507311.

[248] Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. "Software fault proneness prediction using support vector machines". In: *Proceedings of the world congress on engineering*. Vol. 1. 2009, pp. 1–3.

[249] Jeongju Sohn and Shin Yoo. "FLUCCS: Using code and change metrics to improve fault localization". In: *ISSTA 2017 - Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2017), pp. 273–283. DOI: 10.1145/3092703.3092717.

[250] Joengju Sohn and Shin Yoo. "Empirical Evaluation of Fault Localisation Using Code and Change Metrics". In: *IEEE Transactions on Software Engineering* 5589 (c) (2019), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/tse.2019.2930977.

[251] D. Spinellis and M. Jureczo. *CKJM extended - An extended version of Tool for Calculating Chidamber and Kemerer Java Metrics (and many other metrics)*. http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/. (Accessed on 12/16/2020). May 2011.

[252] Statista. *Mobile operating systems' market share worldwide from January 2012 to July 2020*. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. (Accessed on 11/12/2020). Aug. 2020.

[253] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. "ReduKtor: How We Stopped Worrying About Bugs in Kotlin Compiler". In: (2019). arXiv: 1909.07331. URL: http://arxiv.org/abs/1909.07331.

[254] Andrew Sutton, Ryan Holeman, and Jonathan I. Maletic. "Identification of Idiom Usage in C++ Generic Libraries". In: *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*. ICPC '10. USA: IEEE Computer Society, 2010, pp. 160–169. ISBN: 9780769541136. DOI: 10.1109/ICPC.2010.37.

[255] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan. "Predicting Bugs Using Antipatterns". In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 270–279. DOI: 10.1109/ICSM.2013.38.

[256] *Tail Recursive*. Online; accessed 01-July-2019. Apr. 2016. URL: https://kotlinlang.org/docs/reference/functions.html#tail-recursive-functions.

[257] M. Tan, L. Tan, S. Dara, and C. Mayeux. "Online Defect Prediction for Imbalanced Data". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. 2015, pp. 99–108. DOI: 10.1109/ICSE.2015.139.

[258] V. Tankov, Y. Golubev, and T. Bryksin. "Kotless: A Serverless Framework for Kotlin". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 1110–1113.

[259] Andrey A. Terekhov and Chris Verhoef. "Realities of language conversions". In: *IEEE Software* 17 (6) (2000), pp. 111–124. ISSN: 07407459. DOI: 10.1109/52.895180.

[260] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan. "What are the characteristics of high-rated apps? A case study on free Android Applications". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 301–310. DOI: 10.1109/ICSM.2015.7332476.

[261] Yuan Tian, Dinusha Wijedasa, David Lo, and Claire Le Gouesy. "Learning to rank for bug report assignee recommendation". In: *IEEE International Conference on Program Comprehension* 2016-July (2016), pp. 1–10. DOI: 10.1109/ICPC.2016.7503715.

[262] Marco Trudel, Carlo A. Furia, Martin Nordio, Bertrand Meyer, and Manuel Oriol. "C to O-O translation: Beyond the easy stuff". In: *Proceedings - Working Conference on Reverse Engineering, WCRE* (2012), pp. 19–28. ISSN: 10951350. DOI: 10.1109/WCRE.2012.12.

[263] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. "When and Why Your Code Starts to Smell Bad". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 403–414. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818805.

[264] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. "An Empirical Study on the Impact of C++ Lambdas and Programmer Experience". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 760–771. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884849.

[265] Raoul-Gabriel Urma, Dominic Orchard, and Alan Mycroft. "Programming Language Evolution Workshop Report". In: *Proceedings of the 1st Workshop on Programming Language Evolution*. PLE '14. Uppsala, Sweden: Association for Computing Machinery, 2014, pp. 1–3. ISBN: 9781450328876. DOI: 10.1145/2717124.2717125.

[266] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. "Soot - a Java Bytecode Optimization Framework". In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–. URL: http://dl.acm.org/citation.cfm?id=781995.782008.

[267] Milos Vasic. *Mastering Android Development with Kotlin*. Packt Publishing Ltd, 2017.

[268] B. Verhaeghe, A. Etien, N. Anquetil, A. Seriai, L. Deruelle, S. Ducasse, and M. Derras. "GUI Migration using MDE from GWT to Angular 6: An Industrial Case". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019, pp. 579–583.

[269] Daniël Verloop. "Code smells in the mobile applications domain". Master Thesis. Delft University of Technology, 2013. URL: https://repository.tudelft.nl/islandora/object/uuid:bcba7e5b-e898-4e59-b636-234ad3fdc432?collection=education.

[270] Joost Visser. *SIG/TÜViT evaluation criteria trusted product maintainability: Guidance for producers*. https://www.softwareimprovementgroup.com/wp-content/uploads/2020-SIG-TUViT-Evaluation-Criteria-Trusted-Product-Maintainability-Guidance-for-producers.pdf. 2015.

[271] J. Walden, J. Stuckman, and R. Scandariato. "Predicting Vulnerable Components: Software Metrics vs Text Mining". In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, pp. 23–33. DOI: 10.1109/ISSRE.2014.32.

[272] Feng Wang, Jinxiao Huang, and Yutao Ma. "A Top-k Learning to Rank Approach to Cross-Project Software Defect Prediction". In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC* (2018), pp. 335–344. ISSN: 15301362. DOI: 10.1109/APSEC.2018.00048.

[273] S. Wang and X. Yao. "Using Class Imbalance Learning for Software Defect Prediction". In: *IEEE Transactions on Reliability* 62 (2) (2013), pp. 434–443. DOI: 10.1109/TR.2013.2259203.

[274] Anthony I. Wasserman. "Software Engineering Issues for Mobile Application Development". In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER '10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 397–400. ISBN: 9781450304276. DOI: 10.1145/1882362.1882443.

[275] John Waters. *Kotlin Goes Open Source*. https://adtmag.com/articles/2012/02/22/kotlin-goes-open-source.aspx. 2012.

[276] *Will "It" variable stay or go?* Online; accessed 01-July-2019. Jan. 2017. URL: https://discuss.kotlinlang.org/t/will-it-variable-stay-or-go/522/19.

[277] David Winer. *Android Developers Blog: Android's commitment to Kotlin*. https://android-developers.googleblog.com/2019/12/androids-commitment-to-kotlin.html. (Accessed on 10/19/2020). Dec. 2019.

[278] Di Wu, Lin Chen, Yuming Zhou, and Baowen Xu. "An Extensive Empirical Study on C++ Concurrency Constructs". In: *Inf. Softw. Technol.* 76 (C) (Aug. 2016), pp. 1–18. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.04.004.

[279] Jifeng Xuan and Martin Monperrus. "Learning to combine multiple ranking metrics for fault localization". In: *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014* (2014), pp. 191–200. DOI: 10.1109/ICSME.2014.41.

[280] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. "Static Control-Flow Analysis of User-Driven Callbacks in Android Applications". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 89–99. ISBN: 9781479919345.

[281] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. "Deep Learning for Just-in-Time Defect Prediction". In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. Aug. 2015, pp. 17–26. DOI: 10.1109/QRS.2015.14.

[282] Xiaoxing Yang, Ke Tang, and Xin Yao. "A learning-to-rank approach to software defect prediction". In: *IEEE Transactions on Reliability* 64 (1) (2015), pp. 234–246. ISSN: 00189529. DOI: 10.1109/TR.2014.2370891.

[283] Xin Ye, Razvan Bunescu, and Chang Liu. "Learning to rank relevant files for bug reports using domain knowledge". In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* 16-21-November (2014), pp. 689–699. DOI: 10.1145/2635868.2635874.

[284] Xin Ye, Razvan Bunescu, and Chang Liu. "Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation". In: *IEEE Transactions on Software Engineering* 42 (4) (2016), pp. 379–402. ISSN: 00985589. DOI: 10.1109/TSE.2015.2479232.

[285] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. "A New Android Malware Detection Approach Using Bayesian Classification". In: *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*. 2013, pp. 121–128. DOI: 10.1109/AINA.2013.88.

[286] Zhongxing Yu, Chenggang Bai, Lionel Seinturier, and Martin Monperrus. "Characterizing the Usage, Evolution and Impact of Java Annotations in Practice". In: *IEEE TSE: Transactions on Software Engineering* (2019). URL: https://hal.inria.fr/hal-02091516.

[287] Lotfi A Zadeh. "Fuzzy Logic and Its Application to Approximate Reasoning." In: *IFIP Congress*. Vol. 591. 1974.

[288] Harry Zhang. "The Optimality of Naïve Bayes". In: *In FLAIRS2004 conference*. 2004.

[289] Jack Zhang, Shikhar Sagar, and Emad Shihab. "The Evolution of Mobile Apps: An Exploratory Study". In: *Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile*. DeMobile 2013. Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 1–8. ISBN: 9781450323123. DOI: 10.1145/2501553.2501554.

[290] Fei Zhao, Yaming Tang, Yibiao Yang, Hongmin Lu, Yuming Zhou, and Baowen Xu. "Is Learning-to-Rank Cost-Effective in Recommending Relevant Files for Bug Localization?" In: *Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015* (2015), pp. 298–303. DOI: 10.1109/QRS.2015.49.

[291] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. "Mining API mapping for language migration". In: *Proceedings - International Conference on Software Engineering* 1 (2010), pp. 195–204. ISSN: 02705257. DOI: 10.1145/1806799.1806831.

[292] Hui-Juan Zhu, Tong-Hai Jiang, Bo Ma, Zhu-Hong You, Wei-Lei Shi, and Li Cheng. "HEMD: a highly efficient random forest-based malware detection framework for Android". In: *Neural Computing and Applications* 30 (11) (2018), pp. 3353–3361. ISSN: 1433-3058. DOI: 10.1007/s00521-017-2914-y.

[293] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. "Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process". In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '09. Amsterdam, The Netherlands: Association for Computing Machinery, 2009, pp. 91–100. ISBN: 9781605580012. DOI: 10.1145/1595696.1595713.

Appendix A

Publications

Published:

- Bruno Góis Mateus and Matias Martinez. "An empirical study on quality of Android applications written in Kotlin language". In: *Empirical Software Engineering* 24 (6) (2019), pp. 3356–3393. DOI: 10.1007/s10664-019-09727-4
- Bruno Gois Mateus and Matias Martinez. "On the Adoption, Usage and Evolution of Kotlin Features in Android Development". In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM '20. Bari, Italy: Association for Computing Machinery, 2020. ISBN: 9781450375801. DOI: 10.1145/3382494.3410676

To submit:

- Bruno Góis Mateus and Matias Martinez. *"A learning-based recommendation system to support migration of Android applications from Java to Kotlin"*
- Matias Martinez and Bruno Gois Mateus. *How and Why did developers migrate Android Applications from Java to Kotlin? A study based on code analysis and interviews with developers.* 2020. arXiv: 2003.12730 [cs.SE]

Appendix B

Kotlin Features

When a new programming language is released, it offers developers a set of language *features*. Other languages could already provide some of these features, whereas other features can be completely new. In this section, we exclusively focus on features that are available in Kotlin but not in Java. Our goal is to show programming language features that were not fully available for developing Android applications before the release of Kotlin.

To identify these features, we inspected the Kotlin official website. First, we extracted 13 features from a document that compares Kotlin and Java [57]. Then, we extracted 4 features from Kotlin's release notes, which were not mentioned in the comparison document (*coroutine* as experimental feature and *type alias* from release 1.1 and *contract* and *inline class* from release 1.3). Finally, we passed over the Kotlin Reference [129] and identified 7 more features. Table B.1 summarizes the features identified.

B.1 Type inference

Kotlin is a strongly typed language. However, developers do not always need to declare types explicitly. The compiler attempts to infer the type of an expression based on information that it includes. *Type inference* can happen in different situations and the Listing 4 shows: i) when a return type of a function can be inferred, as line 1 displays and ii) when the type of a variable can be inferred based on the assignment expression, shown in line 2.

Other programming languages like Haskell and Swift also have *type inference*.

```
1 fun plusOne(x: Int) = x + 1
2 val listS = listOf("A", "B") // List<String>
```

Listing 4: Kotlin type inference examples

Table B.1: Kotlin features and their release version.

ID	Feature	Release version
1	Type inference	1.0
2	Lambda	1.0
3	Inline function	1.0
4	Null-safety (Safe and Unsafe calls)	1.0
5	When expressions	1.0
6	Function w/arguments with default value	1.0
7	Function w/ named arguments	1.0
8	Smart casts	1.0
9	Data classes	1.0
10	Range expressions	1.0
11	Extension Functions	1.0
12	String template	1.0
13	Delegation (Super and Property delegation)	1.0
14	Operator Overloading	1.0
15	Singleton	1.0
16	Companion Object	1.0
17	Destructuring Declaration	1.0
18	Infix function	1.0
19	Tail-recursive function	1.0
20	Sealed class	1.0
21	Type aliases	1.1
22	Coroutine	1.3[1]
23	Contract	1.3
24	Inline class	1.3

```
button.setOnClickListener { toast("Bye boilerplate") }
```

Listing 5: Example showing how developers could use lambdas to handle a click event on a button in Android developement.

```
inline fun someMethod(a: Int, func: () -> Unit):Int {
    func()
    return 2*a
}
```

Listing 6: Example of inline function in Kotlin.

B.2 Lambda expressions and Anonymous functions

Lambda expressions and *Anonymous functions* are both 'function literals', i.e., functions that are not declared but passed as an expression.[2] Using *Lambda*, developers make the code more succinct and readable by introducing more concise expressions [88]. Furthermore, since the application and the Android platform interact through a variety of callbacks [280], *lambda* functions are useful in Android development, fitting with the use of callback to handle user events. For example, the following listing shows how developers could use lambdas to handle a click event on a button in Android development.

B.3 Inline Function

Using *inline function*, developers can eliminate certain run-time penalties imposed by higher-order functions due to memory allocations and virtual calls. The keyword inline means that a function, as well as function parameters, will be expanded at the call site, consequently helping to reduce call overhead.

The run-time overhead is a consequence of the fact that in Kotlin functions are higher-order functions. Consequently, they are objects, and they capture a closure, i.e., those variables that are accessed in the body of the function.

Declaring a function inline is simple, just add the inline keyword to the function definition:

B.4 Null-safety

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references). If a developer wants to access a property of one object that has no safe guarantee (i.e., could be null), the compiler reports an error. Thus, the developer has three options. First, developers can explicitly check whether an object is null,

[2] https://kotlinlang.org/docs/reference/lambdas.html

```
1 val a = "Kotlin"
2 val b: String? = null
3 println(b?.length)
```

Listing 7: Example of use of safe call to access a property of a nullable variable safely.

as we would usually do in Java. Second, the Kotlin's safe call operator represented by the symbol "?" could be used. In this case, the property is returned if the object is not null, otherwise null is returned. The third option is the not-null assertion operator (!!) that converts any value to a non-null type and throws an exception if the value is null.[3]

The following listing shows two variables declared, but 'b' is a nullable reference. Consequently, it could hold null. To access a property of 'b', it is necessary to use the safe call operator "?" as we can see in line 2. In this case, the snippet returns b.length if b is not null, and null otherwise. Therefore, using this operator, developers might write code less prone to NullPointerException.

However, although Kotlin's type system is aimed to eliminate NullPointerExceptions (NPE), it is possible that Kotlin programs suffer from NPE and the possible causes are: i) An explicit call to throw *NullPointerException*; ii) Usage of the not-null assertion operator '!!'; iii) Java interoperation; iv) Some data inconsistency with regard to initialization.

B.5 When expression

When expressions replace switch operators of C-like languages and provides flexibility: i) if many cases should be handled in the same way, the branch conditions may be combined with a comma, as Listing 8 shows at line 2; ii) developers can use arbitrary expressions (not only constants) as branch conditions; iii) branch conditions can contain an expression that checks a value for being in (*in*) or not in (!*in*) a range or a collection as we can see in Listing 8 at line 4; iv) also, branch conditions can be used to check whether a value is (*is*) or is not (*!is*) of a particular type as shown in line 5. Furthermore, it can be used as an expression. In this case, the value of the satisfied branch becomes the value of the overall expression.[4]

B.6 Default argument

Default arguments allow developers to write methods or constructors with optional arguments. When an optional argument is not passed in the method call, it receives a default value. Default values are defined using the '=' after type along with the value, the same syntax used in Python, as Listing 9 shows.

Thus, instead of writing multiple constructors or overload multiple methods, developers could use default arguments. Furthermore, this feature allows developers to write more concise, more expressive, more readable, and consequently more maintainable code.

[3] https://kotlinlang.org/docs/reference/null-safety.html
[4] https://kotlinlang.org/docs/reference/control-flow.html

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    is Float -> print("is float")
    else -> print("none of the above")
}
```

Listing 8: Example of when expressions with different types of conditions.

```
fun displayGreeting(message: String, name: String = "Guest") {
    println("$message $name")
}
displayGreeting("Welcome", "John") //prints "Welcome John"
displayGreeting("Welcome") //prints "Welcome Guest"
```

Listing 9: Example of function that has one optional argument.

B.7 Named argument

A *named argument* is an argument passed in a method call with its name specified. In that way, developers do not need to follow the same order of arguments found in the method declaration. Moreover, *named arguments* can increase the readability of code, since they provide more information about the arguments on the function call.

In Listing 10, there is one function declared at line 1 that has 4 'Boolean' arguments with a default value. This function is called at line 8 and 9, the last one uses named arguments to improve the source code readability, providing more information about each optional argument.

B.8 Smart cast

Smart casts make it possible to avoid the use of explicit cast operators because the compiler tracks is-checks (*instanceof* equivalent) and explicitly casts for immutable values. Listing 11 shows a function that tests at line 2 whether the variable *x* is String. Then, due to the smart cast mechanism, there is no need to cast *x* to 'String' inside the 'if' body. Therefore, there is no need for writing explicit cast after a type checking using the *instaceof* statement.[5]

[5] https://kotlinlang.org/docs/reference/typecasts.html

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

Listing 10: Example of use of named arguments that increases the source code readability.

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to String
    }
}
```

Listing 11: Example of use of named arguments that increases the source code readability.

```
data class User(val name: String, val age: Int)
```

Listing 12: Example of a data class definition.

```
for (i in 1..4) print(i)
for (i in 4 downTo 1) print(i)
```

Listing 13: Example of range expressions in for loop iteration.

B.9 Data classes

Data classes come to replace the Plain Old Java Object (POJO) and, consequently, its boilerplate code. In Java, to create a POJO, a developer needs to define a class with constructor(s), fields to store data, *getters* and *setters* for each field, equals(), etc. Using *data class*, the Kotlin compiler performs all of this work. As shown in Listing 12, the syntax of a data class is quite simple and concise.

B.10 Range expression

Range expression makes the syntax of *for loop* iteration and control flow statements more readable as Listing 13 shows. It implements a common interface in the library: '*ClosedRange<T>*' that denotes a closed interval in the mathematical sense, defined for comparable types. It has two endpoints: *start* and *endInclusive*, both included in the range. The main operation is *contains*, usually used in the form of 'in'/'!in' operators. Furthermore, they are formed with *rangeTo* functions that have the operator form '..'.[6]

B.11 Extension function

Extension function provides the ability to extend a class with new functionality without having to inherit from that class or use any design pattern such as Decorator. Using special declarations called extensions, it possible to create functions and property extensions. For instance, in Listing 14 a function to swap two elements of a list is added to instances of 'MutableList'.

B.12 String template

String template may contain pieces of code that are evaluated and whose results are concatenated into the 'String' as shown in Listing 15.

[6] https://kotlinlang.org/docs/reference/ranges.html

```kotlin
fun MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'l'
```

Listing 14: Example of extension function that adds a possibility of swap element of 'MutableList'.

```kotlin
val i = 9
println("i = $i") // prints "i = 9"
println("i + 1 = ${i + 1}) // prints "i + 1 = 10"
```

Listing 15: Example of extension function that adds a possibility of swap element of 'MutableList'.

B.13 Delegation

Inheritance in object-oriented programs establishes a strong coupling between classes [139], and for this reason, developers should favor object composition over class inheritance [93]. One approach to reach this goal is to replace inheritance with a delegation [86], which is natively supported by Kotlin. Kotlin also supports delegation at the level of properties. This type of delegation aims to avoid code duplication like the *setters* and *getters* of different properties.[7]

For instance, in Listing 16 the class Derived implements the interface Base by delegating all of its public members to a specified object *b*.

B.14 Destructuring declaration

Destructuring declaration makes it possible to break down an object into multiple variables at once. For instance, it can be used to return multiple values for a function or to traverse a map as shown in Listing 17.[8]

B.15 Operator overloading

Operator overloading allows developers to provide implementations for a predefined set of operators. These operators have fixed symbolic representation (like + or *) and fixed precedence.

[7] https://kotlinlang.org/docs/reference/delegated-properties.html
[8] https://kotlinlang.org/docs/reference/multi-declarations.html

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main(args: Array<String>) {
    val b = BaseImpl(10)
    Derived(b).print() //prints 10
}
```

Listing 16: Example of delegation to replace inheritance.

```
for ((key, value) in map) {
   // do something with the key and the value
}
```

Listing 17: Example of the usage of desctructuring declaration combined with a 'for' to traverse a map.

```kotlin
data class Point(val x: Int, val y: Int)
operator fun Point.unaryMinus() = Point(-x, -y)
val point = Point(10, 20)
println(-point)  // prints "Point(x=-10, y=-20)"
```

Listing 18: Example of operator overloading combined with a extension function.

```kotlin
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
}
```

Listing 19: Example of singleton that extends another class.

In Kotlin, to overload an operator, developers should provide a function defined inside a class or an extension function. Moreover, functions that overload operators need to be marked with the *operator* modifier.[9]

Listing 18 shows an extension function that overloads the operator 'unaryMinus'.

B.16 Singleton

Singleton is a design pattern that restricts the instantiation of a class to one single instance. Like Scala, Kotlin provides an easy way to declare *singletons* using *object declarations* as well. An *object declaration* always has a name following the object keyword, as shown in Listing 19. Just like a variable declaration, an *object declaration* is not an expression, and cannot be used on the right-hand side of an assignment statement.[10]

B.17 Companion Object

Companion Object is an alternative to *static* methods since, unlike Java, Kotlin classes do not support static methods. Declaring a companion object inside a class, developers are able to call its members with the same syntax as calling static methods in Java, using only the class name as a qualifier. However, even though the members of companion objects look like Java static members, at run-time, those are still instances members of real objects.[11] Scala, another JVM programming language, also provide *Companion Object*, but they are slightly different, although

[9] https://kotlinlang.org/docs/reference/operator-overloading.html
[10] https://kotlinlang.org/docs/reference/object-declarations.html
[11] https://kotlinlang.org/docs/tutorials/kotlin-for-py/objects-and-companion-objects.html

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

Listing 20: Implementation of the design pattern Factory using companion object.

```
infix fun Int.add(b : Int) : Int = this + b
val x = 10.add(20)
val y = 10 add 20        // infix call
```

Listing 21: Example of usage of infix function.

they share the basic behavior.[12]

Listing 20 shows how one may implement the design pattern Factory using companion object.

B.18 Infix function

Infix functions are functions that can be called without using the period and brackets. They are marked with the keyword *infix* and must satisfy the following requirements: i) they must be member functions or extension functions; ii) they must have a single parameter; iii) the parameter must not accept a variable number of arguments and must have no default value. *Infix function* can result in code that looks more like a natural language as shown in Listing 21.

B.19 Tail-recursive function

Tail-recursive functions are recursive functions whose function call to itself is the last operation it performs. When these functions are declared using the keyword 'tailrec', the compiler is hinted to replace recursion with an iteration. Consequently, it generates a function with no risk of stack overflow as the stack does not have to keep track of each intermediate recursion state. Listing 22 shows a tail-recursive version of the Fibonacci sequence calculator.

B.20 Sealed class

Sealed classes are used for representing restricted class hierarchies, when a value can have one of the types from a limited set, but cannot have any other type.[13] To declare a sealed class, the

[12] https://docs.scala-lang.org/overviews/scala-book/companion-objects.html
[13] https://kotlinlang.org/docs/reference/sealed-classes.html

```
tailrec fun fibonacci(n: Int, a: BigInteger, b: BigInteger): BigInteger {
    return if (n == 0) a else fibonacci(n-1, b, a+b)
}
```

Listing 22: A tail-recursive version the Fibonacci sequence calculator.

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

Listing 23: Example of sealed class combined with when expressions.

keyword *sealed* should be added to the class declaration. A *sealed class* can have sub-classes, but all of them must be declared in the same file as the sealed class itself. Moreover, a *sealed class* is abstract by itself, and it cannot be instantiated directly and can have abstract members. The key benefit of using *sealed classes* comes into play when developers use them in a when expression as Listing 23 shows. Listing 23 displays one example where the else clause is not needed because all possible cases are already covered in the other branches of the when expression.

B.21 Type alias

Type aliases provide alternative names for existing types. If the type name is too long, one can introduce a different shorter name and use the new one instead as Listing 24 shows. Moreover, they do not introduce new types.

```
typealias NodeSet = Set<Network.Node>
typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

Listing 24: Example of sealed class combined with when expressions.

```
1 fun main() {
2     GlobalScope.launch { // launch a new  coroutine in background and continue
3         delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
4         println("World!") // print after delay
5     }
6     println("Hello,") // main thread continues while coroutine is delayed
7     Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive
8 }
```

Listing 25: Example of a simple coroutine.

B.22 Coroutine

A *coroutine* is an operation that can be suspended without blocking a thread. This feature aims at simplifying asynchronous programming. The usage of *coroutines* is a particular advantage for Android developers since Android is single-threaded by default.

Moreover, Kotlin's concept of suspending a function provides a safer and less error-prone abstraction for asynchronous operations than futures and promises.[14] Listing 25 shows a *coroutine* launched at line 2 that is delayed in 1 000 milliseconds, but that does not block the main thread.

B.23 Contract

Contracts allow a function to explicitly describe its behavior in a way that is understood by the compiler.[15] This feature was released in Kotlin 1.3 as experimental with only two wide classes of cases are supported: i) improving *smart casts* analysis by declaring the relation between a function's call outcome and the values of the passed arguments and ii) improving the variable initialization analysis in the presence of higher-order functions.

Listing 26 shows how a *contract* could be used to improve *smart casts*. The *contract* defined at line 2 tells the compiler that the function returns only if the variable *condition* is true. Therefore, the compiler can assure that after line 7, s is a 'String'.

B.24 Inline class

Inline classes are a new mechanism in Kotlin to avoid allocating runtime objects, which are similar to Scala's *Value classes*. An *inline class* must have a single property initialized in the primary constructor and is declared by placing an 'inline' modifier before the name of the class. At run-time, instances of the inline class will be represented using this single property, avoiding run-time overhead due to additional heap allocations.

For instance in Listing 27, the class 'Password' is declared as 'inline class'. Consequently, at line 2, no instantiation of class 'Password' happens. The variable *securePassword* will contain

[14] https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html
[15] https://kotlinlang.org/docs/reference/whatsnew13.html#contracts

```kotlin
fun require(condition: Boolean) {
    contract { returns() implies condition }
    if (!condition) throw IllegalArgumentException(...)
}

fun foo(s: String?) {
    require(s is String)
    ...

}
```

Listing 26: Example of a contract.

```kotlin
inline class Password(val value: String)
val securePassword = Password("Don't try this in production")
```

Listing 27: Example of a simple coroutine.

just a ‘String’.