Bachelor Thesis

July 5, 2017

Exploiting User Feedback for Automated Android Testing

Toward User-Oriented Testing

Lucas Pelloni

of 18.03.1993, Zurich, Switzerland (13-722-038)

supervised by

Prof. Dr. Harald C. Gall

Dr. Sebastiano Panichella

Giovanni Grano

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Bachelor Thesis

Exploiting User Feedback for Automated Android Testing

Toward User-Oriented Testing

Lucas Pelloni

Bachelor Thesis

Author: Lucas Pelloni, lucas.pelloni@uzh.ch

URL: https://bitbucket.org/sealuzh/becloma

Project period: 08.01.2017 - 08.07.2017

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

“Program testing can be used to show the presence of bugs, but never to show their absence.“

Edsger W. Dijkstra

Acknowledgements

The experience I lived with the implementation and subsequently the writing of my bachelor thesis has undoubtedly been one of the most rewarding academic experiences of my life until now. During these six months, I have learned to know the programming world more in depth, appreciating its challenges and its beauties more and more. However, the realization of my thesis would not have been possible without the help of some people. First, I would like to thank all those people who have contributed directly to the realization of my bachelor thesis. After that, those who indirectly allowed me to reach such an important goal of my life: my bachelors degree.

First of all, I would like to express my deepest gratitude to my thesis advisor, Giovanni Grano. I would like to thank him for the patience he has shown during the past six months, for his guidance and support, for his constructive criticism and for having always encouraged me to achieve my best. I think one of the best things of my thesis was that he always allowed me to experiment and to be creative with the assigned tasks. I was able to "bang my head" against the new challenges, which helped me to find the right solutions. The final result of my thesis would not have been possible without his help. I would like to offer special thanks to Dr. Sebastiano Panichella for believing in me from the beginning and for giving me such a high-level thesis. Thanks for allowing me to discover a branch of computer science that I did not know in such detail before. I would also like to thank my two computer science schoolmates, Ile and Erion, who helped me a lot with their knowledge of android mobile devices. Finally, I would like to thank Professor Gall for giving me the opportunity to work together with the Software Evolution & Architecture Lab at the University of Zurich, allowing me to use the most state-of-the-art infrastructures and technologies for my thesis.

I would like to spend some words for those people, who have supported me during the whole bachelor degree. I would like to thank my parents for being there for me throughout all my bachelor track and for supporting me in times of need. I could never have done it without you and I hope you will be proud of me.

Abstract

In the last years, the massive distribution of mobile devices like smartphones, tablets and more recently wearables, has radically changed our social life. Since the introduction of the first modern smartphone, the iPhone in 2007, we have witnessed a gradual shift from the traditional paradigm in the use of technology, entering the so called *post-pc* era. Nowadays, the mobile market attracts always more developers and software firms. To sustain this fierce competition, they need to build high quality apps and at the same time, reach the market as soon as possible. It comes naturally to note that testing plays an important role in this process. Research focused for decades on traditional testing, aiming at reaching its maximum automation. However, automated testing for mobile applications presents different challenges and limitations that still need to be properly investigated. This thesis work tries to shed some initial light into possible solutions for such problems. In particular, we focused our attention on the knowledge that can be gained from mobile stores. Indeed, such stores represent an enormous amount of data easily available, like user reviews, and are an unmatched opportunity for software engineering research. Our final aim is to demonstrate how such user feedback can be in some way exploited to integrate and complement the state of art Android automated testing tools. Our results show that a noticeable set of problems can be actually detected only through user feedback. Such observation put the first stone down for a new paradigm of *user oriented testing*. We rely on the linking approach developed in this work in order to be able to enrich the generated crash logs with human-readable descriptions elicited from the connected user reviews. Therefore, we envision new generation of tools that are able to learn from such user reviews which are the components of a given mobile application to exercise more in depth, acting a sort of *user-driven* prioritization of the testing effort.

Zusammenfassung

Durch die massive Zunahme der Verbreitung von Mobilfunkgeräten wie Smartphones, Tables und zuletzt Wearables, hast sich in den letzten Jahren unser Sozialverhalten drastisch geändert. Seit der Einführung des ersten "richtigen" Smartphones, des iPhones im Jahre 2007, wurden wir Zeuge eines allmählichen Wechsels von der traditionellen Technologienutzung hin zu dem, was man als "Post-PC-Ära" bezeichnet. Heutzutage wird der Mobilgerätemarkt für Entwickler und Softwarehäuser immer attraktiver. Um in diesem harten Wettbewerb mithalten zu können, müssen diese immer qualitativer hochwertigere Anwendungen (Apps) entwickeln und so schnell wie irgend möglich auf dem Markt anbieten. Von größter Wichtigkeit ist in diesem Prozess das Testing. Die Forschung fokussierte sich seit Jahrzehnten auf Standardtestverfahren mit dem Ziel, eine weitestgehende Automatisierung zu erreichen. Im Hinblick auf das Testing von Mobile Apps kommt dieses althergebrachte Testing aber an seine Grenzen und stellt neue Herausforderungen, die noch eingehend untersucht werden müssen. Diese Arbeit beschäftigt sich mit der Fragestellung, wie derartige Probleme einer Lösung zugeführt werden können. Im besonderen beschäftigen wir uns mit dem Know-how, welches recht einfach über Mobile-Stores verfügbar ist. Die enorme Menge an Daten und Informationen, die über Mobile-Stores frei verfügbar ist, erscheint uns bemerkenswert. Wir sprechen hier z. B. von User-Reviews, die eine bislang unübertroffene Quelle für Software Engineering Research darstellen. Unser Ziel ist es, darzustellen, wie das Feedback der User so ausgewertet werden kann, dass es ein wesentlicher und integraler Bestandteil der automatisierten Android-Testing-Tools wird. Unsere Ergebnisse zeigen beeindruckend auf, dass eine beachtliche Menge von Problemen und Fehlern, nur allein durch das User-Feedabck gelöst werden konnte.Insofern legen diese Beobachtungen den Grundstein für eine neue Anschauung (Paradigma) des sogenannten *User Oriented Testing*. Wir sehen den in dieser Arbeit entwickelten Ansatz als hinreichend valide an, um die erzeugten Crash-Logs durch benutzerbasierte Beschreibungen bereichern zu können, die aus den angeführten User-Reviews generiert wurden. Aus diesem Grunde, stellen wir eine neue Generation von Tools vor, die in der Lage ist, aus solchen zu "lernen", welche die Komponenten einer definierten mobilen App sind. Dies deshalb, weil sie eine Form der *benutzergesteuerten* Priorisierung des Testverfahrens darstellen.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

This thesis is placed in the context of mobile automated testing. In particular, we aim to shed some initial lights into the possibility to integrate user feedback into such process, in order to increase its efficiency and efficacy. The first Chapter of this work gives an overview about the context in which the thesis is placed, as well as the motivations that led to such thesis. There follows a brief introduction about mobile testing, its limitations and challenges. At the end, we disclose the research questions leading our investigations. The remainder of this thesis is organized as follows:

Chapter 2 presents the recent literature in two areas of interest for this work, *i.e.*, automated mobile testing and the exploiting of mobile stores mining in order to facilitate the software maintenance activities.

Chapter 3 describes the approach we developed in order to investigate our main goal.

Chapter 4 presents our experimental tool, which firstly uses some automated testing tools in order to test a set of mobile applications. Secondly, it clusters possible generated logs which refer to the same bugs. Finally, it integrates user feedback into the testing process, linking them to clustered crash reports.

Chapter 5 describes the empirical study we conducted in order to evaluate the aforementioned approach on a set of 3 mobile apps (available on *FDroid* [1]) with their correspondent reviews crawled from the Google Play Store[2].

Finally, Chapter 6 closes the main body of the thesis drawing the necessary conclusions and spreading the main ideas for future contributions.

1.1 Context

Testing is the action of inspecting the behavior of a program, with the intention of finding anomalies or errors [37]. The goal behind software testing is to exercise the system under test (SUT) in order to hopefully reveal as many bugs as possible. In order to do that, it aims to reach the highest *test coverage* with the smallest number of *test cases*. A test case can be viewed as a set of program inputs, each of them gets associated with an expected result (*i.e., oracle*) when they are executed.

Nowadays, software testing is widely recognized as an essential part of any software development process, presenting however an extremely expensive activity. Indeed, testing all combinations of all possible input values classes for an application [19] requires a lot of workforce and it is almost always unthinkable to reach a testing-coverage of 100%. Furthermore, testing is often performed under time and budget constraints [21] and complex applications might have

[1] `https://f-droid.org`

[2] `https://play.google.com/`

a very large number of potential test scenarios, many of which could be really difficult to predict. Indeed, *Dijkstra* [12] observed that testing a software does not imply a demonstration of the right behaviour of the program, but it only aims to demonstrate the presence of faults, not their absence. So, deeply testing a SUT increases the probability that this software will behave as expected also in the untested cases [19].

In general, there are four testing levels a tester should perform in order to investigate the behaviour of a traditional software:

1. Unit testing;
2. Integration testing;
3. System testing;
4. Acceptance testing.

With *unit testing*, the application components are viewed and split into individual *units* of source code, which are normally functions or small methods. Intuitively, one can view a unit as the smallest testable part of an application. This kind of testing is usually associated with a *white-box* approach. *Integration testing* is the activity of finding faults after testing the previous individually tested units combined and tested as a group together. *System testing* is conducted on a complete, integrated system to evaluate the compliance with its requirements. It can be imagined as the last checkpoint before the end customer. Instead, *acceptance testing* (or *customer testing*) is the last level of the testing process, which states whether the application meets the user needs and whether the implemented system works for the user. This kind of testing is usually associated with a *black-box* approach.
These mentioned testing levels shall be sequentially executed and are driven by two testing methodologies [45,48]:

- black-box testing;
- white-box testing.

With *black-box* testing (also called functional testing) the tester does not have any prior knowledge of the internal structure of the SUT. He tests only the functionalities provided by the software without any access to the source code. Typically, when performing a black box test, a tester interacts with the system's user interface by providing inputs, examining the compliance of the output with the oracles. On the other hand, with *white-box* testing (also *glass testing* or *open box testing*), the test cases are extrapolated from the internal software's structure [21]. Indeed, the tester writes the test cases that reflect paths through the code, with the aim usually to maximize a coverage criterion. At the end, those techniques aim to generate a *test suite* composed by a set of *test cases*, *i.e.*, a set of conditions under which a tester will determine whether a software system is working as it was described by its original specification.

As previously explained, testing can be done through different approaches, methodologies and level of details. However, the common goal behind the testing remains the same, i.e. generating the smaller test suiteable to reveal the largest number of failures, in order to increase the reliability of the system [21]. Nevertheless, the process of finding the correct test scenarios, generating the consequential data inputs, executing and finally reporting the results, comparing them to the previously written specifications, might be time-consuming and cost-intensive. In fact, testing costs have been estimated at being at least half of the entire development cost [6]. For this reason, it is necessary to reduce such costs automatizing the various processes occurring in testing, trying to improve the effectiveness of the whole process. In such described scenario, for instance, a remarkable example is the application of Search-Based approaches that have been successfully exploited in the last years in order to automate the test data generation [23].§

1.2 Motivation

In the last years, we have witnessed a huge growth in the usage of mobile software, due to the advent of the *mobile age*. Indeed, nowadays, application running on mobile devices are becoming so widely adopted, so that they have represented a remarkable revolution in the IT sector. In fact, in three years, over 300,000 mobile applications have been developed [35], 149 billions downloaded only in 2016 [42] and 12 million mobile app developers (expected to reach up 14 million in 2020) maintaining them [11].

Because of the importance of the mobile development field nowadays, developers aim to sustain the competition in acquiring and retaining users increasing the quality of their application. Since testing plays a crucial role in achieving such quality, several approaches for automated mobile testing have been recently developed. This because, mobile applications differ from traditional software and so there is also a need for developing new ad-hoc testing techniques. In fact, mobile applications are structured around Graphical User Interface (GUI) events and activities, thus exposing them to new kinds of bugs and leading to a higher defect density compared to traditional desktop and server applications [25]. Furthermore, they are context-aware [35]. This means that each mobile application is aware of the environment in which it is running and exposes a different behavior according to its current context. This has some implications for the testing, because a test case running on a specific context may lead to different results. In fact, bugs related to contextual inputs are quite frequent [35].

For this reason, mobile applications require new specialized and different testing techniques in order to ensure their reliability [35]. Being mobile applications event-based systems used through UI interaction, the GUI testing needs to have a prominent role in app testing. In particular, in this kind of testing, each test case is designed and run in the form of sequences of GUI interaction events, with the aim to state whether an application meets its written requirements.

As traditional testing, GUI testing can be performed either manual or automatic. However, a manual approach would require a lot of programming and may be time-consuming. For this purpose, with the aim to support developers in building high-quality applications, the research community has recently developed novel techniques and tools to automate their testing process [25, 29, 31, 35]. These techniques consist of automatically creating test cases through the generation of UI and system events, which are transmitted to the application under test, with the aim to cause some crashes. Afterwards, if an exception occurred, these tools usually save in log files the correspondent stack traces, along with the sequence of events that led to the exceptions [35]. However, despite a strong evidence for automated testing approaches in verifying GUI application and revealing bugs, these state-of-art tools cannot always achieve a high code coverage [36]. One reason is that current approaches are not suited for generating inputs that require human intelligence (e.g. *"inputs to text boxes that expect valid passwords, or playing and winning a game with a strategy"*) [29]. This inability to replicate the human behaviour, often leaving feature and crash bugs undetected until they are encountered by the users. Furthermore, current solutions generate redundant/random inputs causing incredibly long sequences of events needed to replicate a crash, while most of such inputs could be avoided. In addition, the crash reports that they generate usually lack of contextual information and are difficult to understand and analyze [8, 26]. For such reasons, sometimes a time-consuming manual intervention may be needed for testing appropriately an application [36].
It is worth noting that, some approaches [29, 31] have proposed new solutions to improve the current automated testing tools, with the aim of making GUI-testing less random. In this respect, MacHiry *et al.* [29] elaborated a novel tool called DYNODROID (see Section 2.1), which sensibly selects the most relevant events to the AUT at a precise state and consequently executes them. On the other hand, Mao *et al.* [31] proposed a multi-objective tool which exploits the cycles of an evolutionary algorithm to improve the testing process. However, none of the above mentioned

approaches have integrated user feedback in the testing process of mobile apps, not exploiting the huge advantage, compared to traditional testing, of having a huge amount of information available from mobile stores, which have been introduced as a new selling paradigm for mobile applications.

Indeed, the exponential growth of the mobile stores offers an enormous amount of information and feedbacks from users that researchers and partitioner can use to develop better applications. In fact, these users feedback are playing more and more a paramount role in the development and maintenance of mobile applications. Indeed, recent research focused on how such knowledge might be successfully exploited (Section 2.2). We argue that also the GUI testing could take profit in some way of this information. That is why, in this thesis work we shed some initial lights into the integration of feedbacks extracted from user reviews in the automated testing for mobile applications. In detail, we hypothesize that such integration can be exploited to overcome the aforementioned limitations of state-of-the-art tools for automated testing of mobile apps in the following ways [39]:

1. it can complement the capabilities of automated testing tools, by identifying bugs that they cannot reveal;
2. it can facilitate the diagnosis and reproducibility of bugs (or crashes), since user reviews often describe the actions that led to a failure;
3. it can be used to prioritize bug resolution activities based on the users' requests, possibly helping developers in increasing the ratings of their apps.

In particular, in this work we investigated this first point by developing an approach able to link the user reviews with the stack traces that refer to the same bug. Furthermore, we conducted a preliminary study to understand how complementary are the two sources of information and in particular whether we can use the reviews to identify some problems that the tools miss.

1.3 Motivation Example

Following, we provide a concrete example of how we plan to compare and analyse the two sources of information: user reviews and reports generated by automated mobile applications testing tools. Given app reviews, we first plan to automatically select the ones describing crashes and bugs in features or in UI elements. A good example of useful review is the following:

"Barely works I only installed this for the android sharing function but it usually loads a white screen when I try to use this feature. It's no better than using a browser to view Facebook otherwise.".

The user here complains about the app sharing function that shows a white screen when activated. A report generated by one of available automated testing tools, SAPIENZ [31], includes the stack trace below, where SAPIENZ is able to find a `NullPointerException` while exercising the sharing functionality.

```
java.lang.NullPointerException
  at com.danvelazco.fbwrapper.activity.BaseFacebookWebViewActivity.shareCurrentPage(BaseFacebookWebViewActivity.java:418)
  at com.danvelazco.fbwrapper.FbWrapper.access$700(FbWrapper.java:26)
  at com.danvelazco.fbwrapper.FbWrapper$MenuDrawerButtonListener.onClick(FbWrapper.java:376)
  at android.view.View.performClick(View.java:4438)
  at android.view.View.onKeyUp(View.java:8241)
  ....
  ....
  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:611)
  at dalvik.system.NativeStart.main(Native Method)
```

So, the aim behind this thesis is to investigate the complementarity between the user reviews and the crash reports generated by the automated state-of-art tools described in Section 2.1. The preliminary step into this analysis it to implement a linking procedure of the two sources of information. We argue that this approach can be fruitfully used to support developers in determining which bugs should be addressed first, prioritizing the arising failures. It is worth noticing that such prioritization is directly guided by the users feedback with the aim to maximize their satisfaction and consequentially, the app rating. We argue that this investigation might improve the automated testing process and might help other developers in validating their mobile applications, using BECLOMA, the implemented tool described in the Chapter 4.

1.4 Research questions

In this thesis, we conduct a preliminary study to investigate the first and most important point from the previous list, *i.e., the complementarity between user feedback and the outcomes of automated testing tools*. This research represents a first fundamental step toward the integration of user feedback into the testing process of mobile apps. Thus, we conducted my thesis aiming to answer the following research questions:

- **RQ**$_1$: *Which are the most important tools for automatic testing of Android applications?*
- **RQ**$_2$: *To what extent can we link the defects arose from automated testing tools with user reviews?*
- **RQ**$_3$: *How complementary are the two source of information? Can we leverage on both of them to increase the effectiveness of the testing process?*

The first research question is merely represented by a literature overview to identify which are the state-of-art tools in automated Android testing. To answer such question we presented the main research in this field in Chapter 2, giving an accurate overview of testing tools in general, their pros and cons and their different exploration strategies.

To answer the **RQ**$_2$, we implement a linking procedure which relies on Information Retrieval (IR) techniques and uses an experimental-based threshold in order to state whether a review refers to a specific stack trace. We described the implemented approach in Chapter 3, discussing its precision in Chapter 5. Finally, for our last research question we relied on the aforementioned implemented linking technique to understand and identify whether particular kind of bugs cannot be revealed by current testing tools but can be observed in user reviews (and vice versa). Again, we describe the achieved results in Chapter 5.

Chapter 2

Related Work

In the following two Sections, I will summarize the main related works on *automated testing tools for Android apps* and on *the broadly usage of user reviews from app store in Software maintenance activities*. An overview of the recent research in the field can be found in the survey by Martin *et al.* [32].

2.1 Automated tools for Android Testing

Depending on their exploration strategy, there are in general three approaches for creating a generation of user inputs on a mobile device [10]: *random testing* [20, 29], *systematic testing* [30] and *model-based testing* [2, 9, 28].

Fuzz testing

When test automation does occur, it typically relies on Google's Android MONKEY command-line [20]. Since it comes directly integrated in Android Studio, the standard IDE for Android Development, it is regarded as the current state-of-practice [30]. This tool simply generates, for the specified Android applications, pseudo-random streams of user events into the system, with the goal of stressing the *AUT*[1] [20]. The effort required for using MONKEY is very low [10]. Users have to specify in the command-line the type and the number of the UI events they want to generate and in addition they can establish the verbosity level of the MONKEY log. The set of possible MONKEY parameters can be found in the official *User Guide* for MONKEY on [20].
The kind of testing implemented by MONKEY follows a black-box approach. Despite the robustness, the user friendliness [10, 29] and the capacity to find out new bugs outside the stated scenarios [1], this tool may be inefficient if the *AUT* requires some human intelligence (*e.g.* a login field) for providing sensible inputs [29].
For this reason, MONKEY may cause highly redundant and senseless user events. Even though it would find out a new bug for a given app, the steps for reproducing it may be very difficult to follow, also due to the randomness in the testing strategy implemented by it [1].

DYNODROID [29] is also a random-based testing approach. However, this tool has been discovered being more efficient than MONKEY in the exploration process [10].
One of the reasons behind a better efficacy has been that DYNODROID is able to generate both *UI events* and *system events* (unlike MONKEY , which can only generate UI events) [10].
Indeed, DYNODROID can simulate an incoming SMS message on a mobile device, a notification of another app or an request of use for available wifi networks in the neighborhood [29]. All these

[1]Application Under Test

Table 2.1: Statistics on found crashes from the SAPIENZ experiment

App crashes	**Monkey**	**Dynodroid**	**Sapienz**
Nr. App crashes	24	13	41
Nr. Unique crashes	41	13	104
Nr. Total crashes	1'196	125	6'866

events represent *non-UI events* and they are often unpredictable and therefore difficult to simulate in a suitable context. DYNODROID views the AUT as an event-driven program and follows a cyclical mechanism, also known as the *observe-select-execute* cycle [29]. First of all, it *observes* which events are relevant to the AUT in the current state, grouping they together (an event must be considered relevant if it triggers a part of code which is part of the AUT). After that, it *selects* one of the previously observed events with a randomized algorithm [10, 29] and finally *executes* it. After the execution of that event the AUT reaches a new state and the cycle stars again.
Another advantage of DYNODROID compared to MONKEY is that it allows users to interact in the testing process providing UI inputs. In doing so, DYNODROID is able to exploit the benefits of combining automated with manual testing [29].

Systematic testing

The tools using a systematic explorations strategy rely on more sophisticated techniques, such as symbolic execution and evolutionary algorithms [10].

SAPIENZ [31] introduced a new Pareto multi-objective search-based technique (the first one in Android testing) to simultaneously maximize coverage and fault revelation, while minimizing the sequence lengths. Its approach combines the above mentioned random-based techniques with a new systematic exploration. SAPIENZ starts its work-flow by instrumenting the AUT, which can be achieved using a *white-box* approach, if the source code is available. Otherwise the "instrumenting-process" follows a *black-box* approach in case only the APK[2] is given. Afterwards, SAPIENZ extracts string constants after an accurate analysis of the APK. These strings are used as inputs for inserting realistic strings into the AUT, which has been demonstrated to improve the testing efficiency [31]. After that, UI events are generated and executed by an internal component called *MotifCore*. This component, as highlighted before, combines random-based testing approaches with systematic exploration strategies. When it gets invoked, it initialises an *initial population* via *MotifCore's Test Generator* and starts the genetic evolution process. During this process, another component called *Test Replayer* manages genetic individuals during the evaluation process of the individual fitness. After that, the individual test cases are translated into Android events by the *Gene Interpreter*, which is in charge of communicating with the mobile device. Another component called *States Logger* checks whether the tests cases find some faults and produces statistics for another component, called *Fitness Extractor*, which computes the fitness.
Finally, reports containing data about testing results, such as code coverage, stack traces and "steps for reproducing the faults" are generated and stored at the end of the process.

The experiment results published on [31] show that SAPIENZ is an out-performer in the automated mobile testing area. Table 2.1 compares the above mentioned tools MONKEY and DYNODROID with SAPIENZ , illustrating the strength of the SAPIENZ approach.

As illustrated in Table 2.1, SAPIENZ found from a set of 68 benchmark apps, 104 unique crashes (while MONKEY 41 and DYNODROID 13).

[2]Android Package Kit, extension used by Android OS for installing a mobile app on an android device

Model-based testing

Model-based tools for testing Android applications are quite popular [31]. Most of these tools [2,9,28,33] generate UI events from models which are either manually designed or created from XML configuration files [31].
For example, *SwiftHand* [9] uses a machine learning algorithm to learn a model of the current *AUT*. This final state machine model [10] generates UI events and due their execution the app reaches new unexplored states. After that, it exploits the execution of these events to adapt and refine the model [9]. *SwiftHand*, in a similar way to MONKEY generates only touching and scrolling UI events and is not able to generate system events [10].

2.2 Usage of users reviews in Software maintenance activities

The concept of app store mining was first introduced by *Harman et al.* [22]. In this respect, they presented an empirical analysis of relationships between three main areas of interest in the context of user reviews: technical, customer-focussed and business. Their approach consists in (i) extracting raw data from the app store (ii) parsing that data with the aim to retrieve all attributes concerning the price, the ratings and descriptions of the app itself (iii) using data mining to extract feature information from the above parsed descriptions and finally (iv) providing statistics about the above mentioned areas of interest.

In the context of app store mining, many researchers focused on the analysis of user reviews to support the maintenance and evolution of mobile applications [32]. Indeed, the survey by *Martin et al.* [32] proposes a study which aims at defining and grouping together different literature sources, demonstrating that an area of research in the field of "App Store Analysis" actually exists.

In this field, Pagano *et al.* [38] empirically analyzed the feedback content and their impact on the user community. Indeed, they proposed a study proposition where (i) they hypothesized that developers find user feedback a useful source of information for validating their software. Then, (ii) they assumed that the current practice of user involvement in the validation process brings with it several challenges, since it is unsystematic and there are some practical limitations concerning the quality, content and structure of such user feedback. Finally, (iii) they believe that developers embrace tool support for dealing with a very large amount of user reviews, which allow them to validate their software according to its users.

In this respect, Khalid *et al.* [27] conducted a study on iOS apps discovering 12 types of user complaints posted in reviews. In their findings, they found that the most frequent regard functional errors, requests about a new feature and complaints about app crashes. With this study, they argue that the analysis of these complaints can further help developers to identify app bugs and develop new features according to their users requests. Furthermore, they argue that based on their findings developers can better anticipate possible complaints and so prioritize them a priori.

Several approaches have been proposed with the aim of classifying useful reviews. AR-MINER [8] represents the first approach proposed in the literature able to discern informative reviews. In this approach, they aim to retrieve the most "informative" information which can be extracted from raw user reviews. They argue, that this kind of information can help developers in app markets to improve their applications. In this sense, *AR-Miner* can be viewed as a "novel analytic framework" [8], which consists of a new ranking scheme that prioritizes these "informative" reviews.

In the context of the usage of users reviews in the validation process of a software, in a recent work Panichella *et al.* introduced a tool called SURF (Summarizer of User Reviews Feedback), that is able to analyse the useful information contained in app reviews and to perform a systematic summarisation of thousands of user reviews through the generation of an interactive agenda of recommended software changes [41].

Finally, Palomba *et al.* [40] proposed CHANGEADVISOR, a tool able to suggest the source code artefacts to maintain according to user feedback. Indeed, this tool is able to analyse a very large amount of user reviews, by automatically (i) extracting these user reviews (ii) grouping them together using some bucketing techniques (iii) linking these source code components relevant to the currently investigated user review.

2.3 Choice and motivation for the used automated testing tools

For the implementation of BECLOMA, we have chosen to use the following two automated testing tools:

- MONKEY
- SAPIENZ

The selection of MONKEY is justified by the fact that (i) MONKEY represents the current state-of-practice in the field of automated mobile testing [30] and therefore is well supported and documented by the community, (ii) the effort in using it is very low [10] and (iii) being a command-line tool, it can easily be combined with the terminal and so be launched by and integrated into an external tool which provides and supports command-line functions.

The aim of selecting SAPIENZ is driven by the fact that (i) SAPIENZ has been found to be (based on the experimental results presented on [31]) an outperformer in the automated mobile testing area. Indeed its approach has been discovered to be much more reliable and high-performing compared to other testing strategies such as those of MONKEY or DYNODROID. (ii) The source code of SAPIENZ is readily accessible on *GitHub* and therefore the whole source code can be integrated into another tool (iii) SAPIENZ can also be started using some command-lines.

Chapter 3

Approach

In this work, we aim to build an approach able to (i) automatically test Android applications with modern state-of-art tools, (ii) automatically collect the arising errors (*i.e.*, stack traces), (iii) detect the unique errors (*i.e.*, the ones that derive from the same bug), (iv) link such stack traces with the user reviews that claim about the correspondent problems. We called this approach **BECLOMA** (**B**ug **E**xtractor, **C**lassifier and **L**inker **O**f **M**obile **A**pps). Figure 3.1 depicts the main actions performed by this approach. To give a clearer and more understandable explanation of how BECLOMA works, we will describe in the following Chapter its key features (see figure 3.1). Such features represent the three main processes that BECLOMA sequentially performs.

1. the TESTING part uses MONKEY and SAPIENZ to test the APKs under test, reporting their testing results and extracting possible *crashes* from the before generated test logs;

2. the CLUSTERING part investigates the similarity between the previous extracted crash logs, using different metrics and strategies, in order to collect them together and create a crash log *bucket*, *i.e.*, a set of unique;

3. the LINKING part represents the core feature of BECLOMA. It preprocesses a set of given *user reviews* as well as the set of the previously created crash logs, in order to preprocess them for the linking procedure. Afterwards, it investigates whether there exists a correlation between the stack traces and the user feedbacks, with the aim to link, if possible, the reviews with the crash logs.

3.1 Testing and Traces Collection

The first step in the overall approach basically relies on two of the most used Android testing tools, in order to exercise the SUT under tests and collect as many failures as possible. First of all, BECLOMA acts as crawler in order to download the set of desired APKs from the *FDroid* API. Thus, it firstly reads a static structured file containing a set of android package names; then, it builds the necessary *HTTP links* in order to download the corresponding APK files. Once the dataset has been built and some testing parameters have been specified, the testing session can be started. The set of parameters that must be specified are described in detail in Section 4.4.4. The overall approach of BECLOMA consists in three testing cycles:

1. The **single app** cycle concerning the testing of a single APK. It represents the time frame for which each APK in the dataset is tested. After that time frame, the approach starts with the testing of the next APK indicated in the dataset.

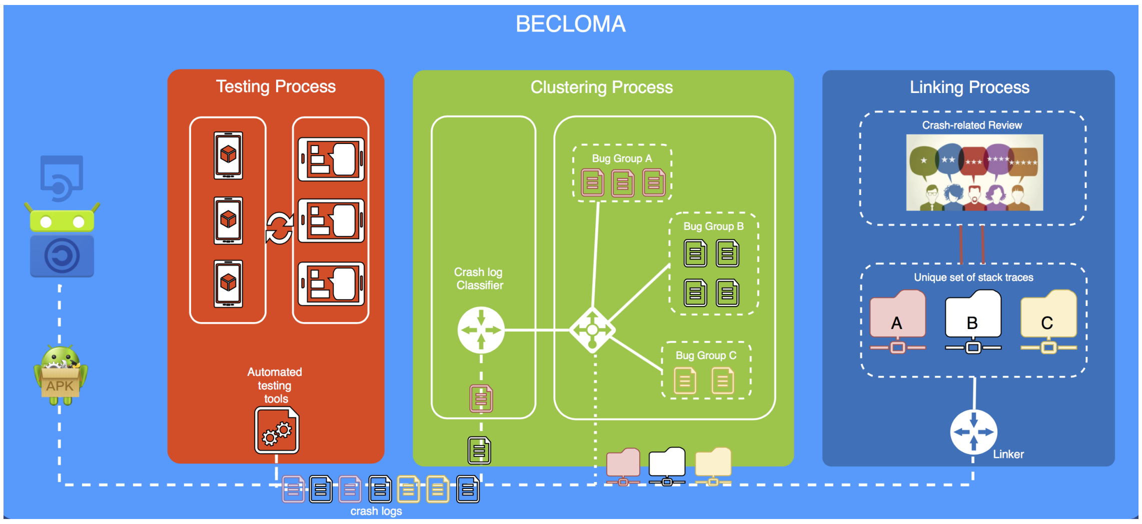

Figure 3.1: BECLOMA approach

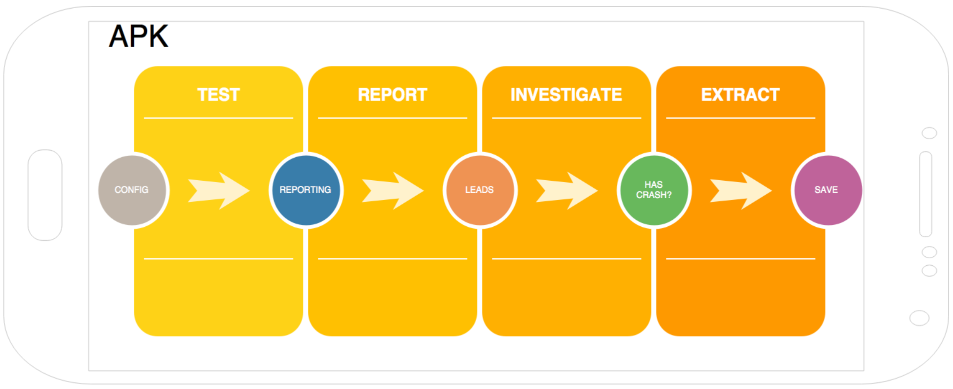

Figure 3.2: Approach performed by BECLOMA for testing a single Android app

2. The **dataset** cycle describing the time spent for testing the APKs within the dataset exactly one time for that currently cycle.

3. The **session** cycle characterizing a testing session, *i.e.*, how many dataset cycles have to be performed.

Figure 3.3 shows the roles of these cycles in the testing approach, while figure 3.2 illustrates the process in detail behind the *single app cycle*. First, during this cycle the APK is systematically *tested* according to its specifications and testing parameters. Once the test is finished, the *reporting* phase starts. This step consists of saving the generated reports in a given directory. Once the logs have been stored, they are ready to be *investigated*. This means, that the reports are parsed with the intention to find eventually crashes inside them. Afterwards, if a crash occurred, its part relative to the stack trace from the whole corpus of the log is *extracted*. At the end, this information is saved into a specific directory aiming at collecting all the arose failures.

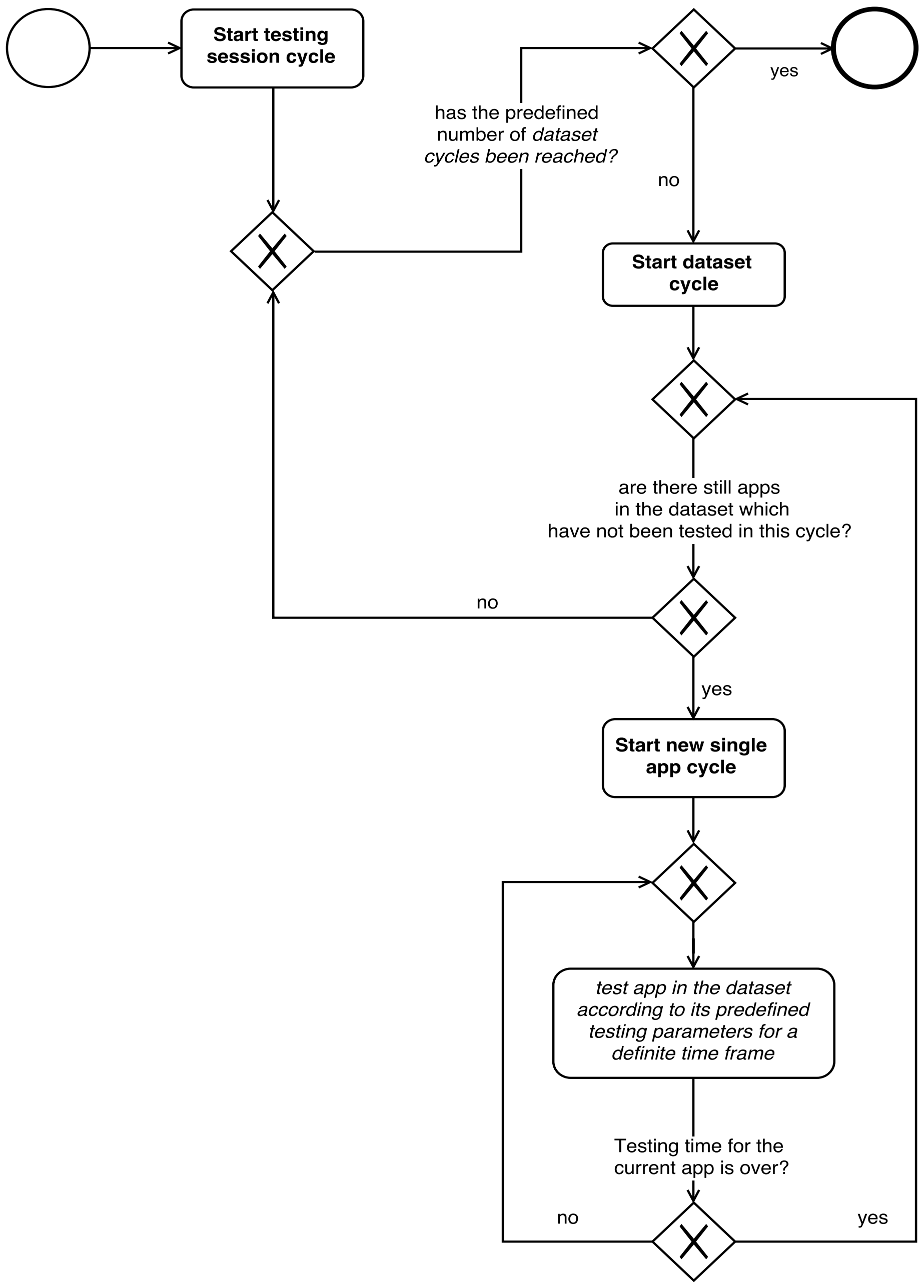
Start testing session cycle
has the predefined number of *dataset cycles been reached?*
yes
no
Start dataset cycle
are there still apps in the dataset which have not been tested in this cycle?
no
yes
Start new single app cycle
test app in the dataset according to its predefined testing parameters for a definite time frame
Testing time for the current app is over?
no
yes

Figure 3.3: Testing cycles characterizing the testing approach

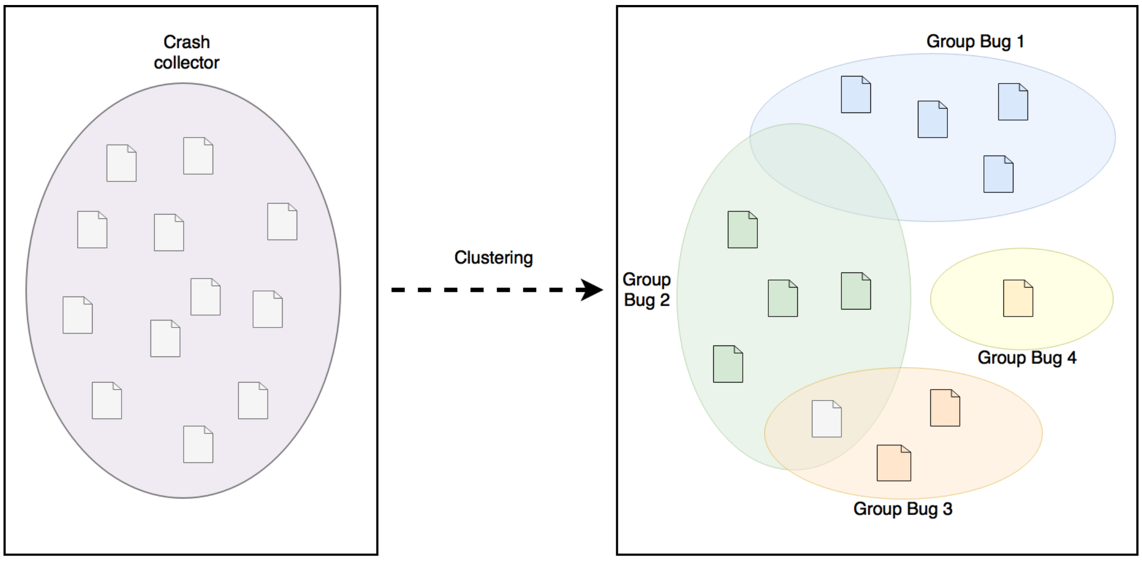

Figure 3.4: The idea behind the clustering process

3.2 Clustering of the Collected Traces

As we said before, at the end of the testing phase all the stack traces that have been generated are stored in a specific directory. However, it is worth noticing that with a high change, most part of such failures would refer to the same bug. To make our explanation better understandable, we introduce a new term we call *unique crashes*. In this sense, a set of unique crashes indicates a group of duplicates in the cluster which belong to the same bug. In fact, each crash in that cluster is considered to be interchangeable.

Indeed, in order to identify these unique crashes, the second step of our approach performs an automatic clustering of the overall bunch of logs. Ideally, at the end of such phase, each cluster should contain logs that refers to exactly the same bug. For this task, we rely on classic Information Retrieval (IR) techniques, collecting features about the logs and comparing them using the *cosine similarity* metric. Figure 3.4 shows the overall process for this task. As said before, some track traces may be overlapping *i.e.*, refers to the same bug of be even duplicates. Despite the trigger method, *i.e.*, the method that raised to the exception, may be the same, there may be different sequences of function calls in the stack trace, the more the analysis goes deep. However, they are hardly detectable and is difficult to affirm that two stack traces which have the same trigger method refer to different bugs. Therefore, some reports may belong to two different bug groups in the bucket. In order to understand better the clustering approach, a clarification of how a crash log is structured must be done. For this purpose, the real structure of a crash report is illustrated in Listing 3.1.

```
// CRASH: com.ringdroid (pid 6207)
// Short Msg: android.database.StaleDataException
// Long Msg: android.database.StaleDataException: Attempted to access a cursor after it has been closed.
// android.database.StaleDataException: Attempted to access a cursor after it has been closed.
//  at android.database.BulkCursorToCursorAdaptor.throwIfCursorIsClosed(BulkCursorToCursorAdaptor.java:64)
//  at android.database.BulkCursorToCursorAdaptor.getCount(BulkCursorToCursorAdaptor.java:70)
...
```

Listing 3.1: Structure of a crash log

A crash log is usually structured as follows:

- *Line 1* represents the top of the crash log, where the concerned package name is made explicit;
- *Line 2* tells in few words the cause of the exception;
- *Line 3* complements the cause of the exception giving a long explanation about the exception itself;
- *Line 4* represents the first line of the stack trace and contains the name and the generic cause of the exception. From this point, all the function calls underlying are part of the stack trace;
- *Line 5* is considered the exact reason for the exception, *i.e.*, the trigger method within the source code that caused the crash;
- From *line 6* moving gradually down until the end of the stack trace, there are other nested function calls which contain additional information about the cause of the exception. Usually, the most important ones for identifying the cause are in the first few lines.

In order to state whether two crash reports refer to the same bug or not, we used a search based approach to identify the correct threshold. Indeed, we built an *Oracle* which is in charge of comparing two crash logs and is able to answer the following question: *Do they refer to the same bug?*. For its answers it makes use of a similarity tolerance, which has been adapted on the base of our experimental results. In fact, we manually created the bucket of unique crash logs and consequently adapted the threshold in the oracle in order to enable it to rightly answer its questions and so reproduce the same bucket.

Preprocessing. The first step of the clustering approach consists in *preprocessing* the crash reports in order to prepare them to be compared to each other. The preprocessing step is fundamental to be performed since (i) it removes possible noise from the logs (remove unless symbols or numbers), (ii) standardizes them making their contents consistent to each other and (iii) parses and converts them identifying different key elements. To perform such phase, the approach follows a grammar-based tokenization technique implemented in the well-know **Apache Lucene** [13] library. In this respect, we collected all words contained in the crash reports and preprocessed them using a Lucene tokenizer, called *Standard Tokenizer* [**?**]. This tokenizer simply splits the word fields into lexical units using punctuation and whitespaces as split points. In addition, it removes unnecessary symbols (*e.g.*, `"//"`). BECLOMA extends the Lucene tokenizer to a further CamelCase regular expression. This because it is a worldwide convention that developers use CamelCase notation for writing programming words such as names of classes, names of functions or names of variables. Since most of the words included in the crash logs are programming language keywords, BECLOMA complements such StandardTokenizer, so that CamelCase text fields can be also split at the upper-case letters into separate words.

TF-IDF. Once all words inside the crash logs have been tokenized, we relied on some well know Information Retrieval techniques in order to link together the duplicated crashes. As features for such process, we relied on the computation of the tf-idf *(term frequency inverse document frequency)* values on the collected stack traces. In this direction, BECLOMA computes tf-idf numerical statistics in order to find the relevance of a word in a collection of documents (in our case, the crash logs). Tf-idf is a well-know term-weighting scheme and usually is used in Information Retrieval or text mining [47]. The importance of a term is given by the number of times it occurs in a particular document, inversely proportional to its appearance in the entire documents collection [7]. Generally, a tf-idf scheme consists of three main components [17]:

- **TF (Term Frequency)**, *i.e.*, how many times a term appears in the currently scored document, where repeated terms indicate the topic of the document; A high *tf* means that the word in question has a high relevance for the document. The following simplified equation [47] describes a formula for calculating it:

$$tf_{x,y} = \frac{n_{x,y}}{\mid d_y \mid}$$

where $n_{x,y}$ represents the number of occurrences of the term t_x in the document d_y, while $\mid d_y \mid$ represents the number of words inside the document d_y.

- **IDF (Inverse Document Frequency)**, *i.e.*, the inverse of the document frequency, that represents the number of document in which the term appears. If the same term appears in fewer documents, *idf* shows a high value, whereas if a term is very common it returns a low value. The equation [47] below describes a simplified version of the inverse document frequency formula:

$$idf_x = \frac{\mid D \mid}{\mid \{d : t_i \in d\} \mid}$$

where $\mid D \mid$ is the number of documents in the collection, while $\mid \{d : t_i \in d\} \mid$ represents the number of documents which contains the term t_i

- TF-IDF, *i.e.*, the product of the two above terms. If it has a high value means that the currently scored term has a high relevance, otherwise if it returns a low value, the term has little relevance.

$$tfidf_{x,y} = tf_{x,y} * idf_x$$

The *idf* metric actually measures how important a term is. This because a term which appears very often in a single document will have a high *tf* score but if this term rarely occurs in the other ones it will also have have a high *idf* score and so a low tf-idf value. This would imply that it shall not have a high relevance. Otherwise, if a word occurs very often both in a single document and in the entire collection it has a high *tf* score and a low *idf* value, which results in a high tf-idf score.
At the end of this phase, BECLOMA has created for each crash log its correspondent vector space model, *i.e.*, a weighted vector term, where each term indicates a new dimension in the vector and is associated with its correspondent tf-idf value.

Cosine Similarity. To state whether two crash reports refer to the same bug, the approach computes cosine similarity between their previously created vectors space model. The cosine similarity is just a measure of similarity between two vectors [46] (in our case, two normalized weighted vectors consisting of their tf-idf scores). Usually, the resulting similarity ranges from -1 to 1, but in the case of Information Retrieval, since the frequency of the terms are always positive, the returned values range from 0 to 1, where 0 indicates that two documents are completely decorrelated, while 1 means that the words contained inside them are exactly the same. The equation describing the cosine similarity between two vectors is as follows:

$$cosine\ similarity = \cos(\theta) = \frac{A \cdot B}{||A||\,||B||}$$

where, in our case A and B are two normalized weighted term vectors consisting of tf-idf values. With the term "normalized" is understood that when two weighted vectors are used to compute cosine similarity among them, for each time a word is contained within a vector but not in the other, that term is inserted into the vector that does not contain it by associating a tf-idf score of 0. Furthermore, in doing so the two vectors will have the same length by making the computation of their dot product possible.

3.3 Linking approach

To study the correlation between user reviews and the outcomes of automated testing tools, BECLOMA assumes that the set of users feedback have been already classified in according to a defined taxonomy and preprocessed. In order to classify the users feedback according to a given taxonomy (and ad oc preprocessed) we relied on an approach implemented by Palomba *et al.* [40]. In this approach, they proposed some machine learning techniques for automatically classifying a set of user reviews according to a defined taxonomy. Furthermore, it performs a systematic Information Retrieval (IR) preprocessing [5] on both the user reviews and *augmented* stack traces aimed at (i) correcting mistakes, (ii) expanding contractions (e.g., *can't* is replaced with *can not*), (iii) filtering nouns and verbs, (iv) removing common words or programming language keywords, and (v) stemming words (e.g., *aiming* is replaced with *aim*).
The text below depicts an example of Information Retrieval preprocessing applied on a user feedback:

"Crashes on Messages I would give this 5 stars but it crashes every time I try to access my messages in the app. I have removed and reinstalled the app signed in and out even reformatted my phone. But it still crashes when I click Messages every time without fail.".

Following, the review is preprocessed applying the techniques described above:

"crash messag i would give 5 star crash everi time i tri access messag app i remov reinstal app sign even reformat phone but still crash i click messag everi time without fail".

Linking between Crash Logs and Reviews The last step of the approach consists of linking the information from the stack traces contained in the reports to the relevant user reviews. However, linking the two sources of information is not at all obvious. This because they come from different worlds: user reviews contain natural human language which describe the overall scenario that led to a failure [34], while the stack traces contain technical information about the exceptions raised during the execution of a certain test case. To account for this aspect, the approach first removes all information that creates noise in the collected stack traces, only the name and cause of the raised exceptions are selected, *i.e.*, the first line of the stack trace (in the example 3.1 the *line 4*). The choice of considering only some specific parts of the stack traces was driven by experimental results. These results illustrated how the linking accuracy was influenced by the presence/absence of this information. After cleaning the reports, the remaining text is *augmented* with the source code methods included in the stack trace. This concretely means that each method extracted from the stack trace gets compared with all methods included in the source code. If a correlation among them exist, the stack trace is *augmented* with the body and the set of words related to the that method. This step extends the information from the reports with contextual information from the source code, possibly providing additional information useful for the linking process. Also in this case, the choice was not random but driven by the experimental results. Afterwards, the approach performs the systematic Information Retrieval (IR) preprocessing [5] used by the approach implemented by *Palomba et al.* Finally, the resulting documents are linked using the asymmetric Dice similarity coefficient [5], which is defined as follows:

$$Dice(review_j, crash_i) = \frac{|W_{review_j} \cap W_{crash_i}|}{min(|W_{review_j}|, |W_{crash_i}|)}$$

where W_{review_j} represents the set of words composing a user review j, W_{crash_i} is the set of words contained in an augmented stack trace i and the min function normalizes the Dice score with respect to the number of words contained in the shortest document between j and i. The asymmetric Dice similarity returns values between [0,1]. In my thesis, pairs of documents having a Dice score higher than **0.5** were considered as linked by the approach.

Chapter 4

Bug Extractor, Classifier and Linker Of Mobile Apps

This Section presents BECLOMA[1], a *Java-based* tool aimed at helping developers during the testing process of their Android mobile applications. The tool is composed by a set of sub-components that operate sequentially. The first one is the component that is charge of exercising the SUT, revealing and collecting crashes. The second component processes the gathered data, as explained in Section 3.2. At the end, the tool is able to recommend traceability links between the stack traces and a set of mined user reviews for the same app under test. Chapter continues with a technical description of the implemented components, as well with the instructions to run the tool.

4.1 Testing Component

First of all, if no set of APKs is available yet, BECLOMA can be exploited for downloading the needed mobile applications from the *F-Droid API*[2]. In this respect, as shown in the picture 4.1, the component FDROIDCRAWLER, is first in charge of parsing a static structured file (*e.g.* a *csv* file) that contains a set of Android packages names that univocally represent a given app of the Google Play Store. The path of this file is given in the CONFIGURATIONMANAGER, which contains a set of static properties that get elaborated by BECLOMA.
Second, FDROIDCRAWLER searches and then extracts a set of *HTTP links* for those android packages that have been found on the API. Afterwards, it builds the correct *HTTP requests* and finally starts the downloading process, saving the returned APKs in a given directory.

The first step of the testing part was to build a set of APKs, with which to perform the testing process. As said, this can be achieved using either FDROIDCRAWLER or can also be manually created. Now, the second step is to prepare and configure the testing environment. In this direction, figure 4.1 shows an example of a simplified set of parameters which must be given a priori in the CONFIGURATIONMANAGER file in order to launch a testing session. Indeed, *lines 22-41* illustrate how the testing parameters concerning MONKEY and SAPIENZ are specified. Furthermore, in this file the directories containing the dataset must be also inserted. These include the location of the static structured file and the directory where the APKs are going to be stored (*lines 4-5*). In addition to them, the directories on which the generated test logs are going to be saved (*lines 10-11*) must be given.
An in-depth explanation about the specifications in the CONFIGURATIONMANAGER can be found in Section 4.4.4.

[1]the source code of BECLOMA is available at `https://bitbucket.org/sealuzh/becloma`
[2]`https://f-droid.org/repo/`

```
 1 /**
 2  * Dataset directories
 3  */
 4 CSV_FILE_DIR = Dataset/dataset.csv
 5 APK_DIR = Downloads/apks
 6 
 7 /**
 8  * Test logs directories
 9  */
10 MONKEY_DIR = Reports/MonkeyReports
11 SAPIENZ_DIR = Reports/SapienzReports
12 
13 /**
14  * Testing session specifications
15  */
16 MINUTES_PER_APP = 30
17 NR_OF_ITERATIONS = 5
18 
19 /**
20  * Monkey parameters
21  */
22 LOG_VERBOSITY = -v
23 PACKAGE_ALLOWED = -p
24 NR_INJECTED_EVENTS = 5000
25 DELAY_BETWEEN_EVENTS = 10
26 PERCENTAGE_TOUCH_EVENTS = 15
27 PERCENTAGE_SYSTEM_EVENTS = 15
28 PERCENTAGE_MOTION_EVENTS = 15
29 IGNORE_CRASH = True
30 
31 /**
32  * Sapienz parameters
33  */
34 SEQUENCE_LENGTH_MIN = 20
35 SEQUENCE_LENGTH_MAX = 500
36 SUITE_SIZE = 5
37 POPULATION_SIZE = 50
38 OFFSPRING_SIZE = 50
39 GENERATION = 100
40 CXPB = 0.7
41 MUTPB = 0.3
```

Listing 4.1: Properties which get elaborated during the testing sessions

As shown in the figure, the tool accepts two further parameters that tune the duration and the execution of the tests *(lines 16-17)*:

- MINUTES_PER_APP, specifies how many minutes an app will be tested. After that time frame, a time-out occurs and the testing process gets restarted with the next app. Referring to the approach explained in Section 3.1, this value represents the *single app* testing cycle.

- NR_OF_ITERATIONS, specifies how many times the whole dataset will be tested. This value, in turn, describes the *session cycle*.

According to the example 4.1 above and assuming that the dataset consists in 10 apps, the total estimated testing time for an entire testing session would be:

$$30\,min\,\frac{per}{app} * 10\,apps * 5\,iterations = 1500\,min.\,(25\,hours).$$

Once the testing environment has been configured, the automated tool with whom the testing is going to be performed must be made explicit. Indeed, it has to be specified as parameter in the MAIN *args* (as mentioned before in Section 2.3, the tool which can be selected is either MONKEY or SAPIENZ).
The last configuration step is to define on which kind of device (*i.e*, a real device, such as a *tablet* or a virtual device, such as an *emulator*) the testing is going to be performed. In addition to them, an additional argument that starts a timer for a better overview during the testing process can also be passed as main argument. BECLOMA supports different types of emulators or real devices running on different android API levels. However, in order to correctly execute SAPIENZ , the API level shall be *Android 4.4, KitKat*.
Listing below shows an example of a possible combination of parameters that could be given as main arguments.

```
$ java -jar BECLOMA.jar -device -monkey -timer
```

Listing 4.2: Possible BECLOMA command-line

Section 4.4.4 describes the full usage of BECLOMA.

Session cycle. Once the configuration phase is terminated, BECLOMA is able to start concrete the testing process. As shown in the picture 4.1, it manages the component SESSIONLAUNCHER, which is in charge of translating the previously specified testing properties into "Java readable code" and initializing the *session cycle* mentioned in the approach. Concretely, after BECLOMA invokes SESSIONLAUNCHER all attached devices respectively the chosen emulators get initialized, *i.e.*, they get rebooted and restarted as root, so that some important write-read-permissions are enabled during the testing session. Whether the timer has been given as main argument, it gets also started.
Once the initialization step has been completed, SESSIONLAUNCHER invokes the APPTESTER component which is in charge of starting the *dataset cycle*. Listing 4.3 shows a simplified code snippet about the beginning of this testing cycle.
First of all, as shown in Listing 4.3, the total number of iterations specified in the CONFIGURATIONMANAGER is read and stored into a constant of type *int* at *line 3*. After that, the specified device gets initialized, statically invoking the correspondent method (*line 4-9*). According to the boolean variable *isTimer*, a timer may also be started (*line 11-13*). Afterwards, a for-loop starts, where at each iteration the method *testAllApp()* gets invoked. Referring to the approach explained in Section 3.1, at each iteration a new object of type APPTESTER is created, which represents exactly one *dataset cycle*. As shown in the figure 4.1, the SESSIONLAUNCHER would be able to instantiate infinite times the class APPTESTER. However, it must create at least one object of that type in order to start a testing session.

```java
private appTester;
public void startTestingSession() throws Exception {
        final int NUMBER_ITERATIONS = ConfigurationManager.
    getNumberOfIterations();
        if (IS_EMULATOR) {
            SessionLauncher.initialiseEmulator();
        }
        else {
            SessionLauncher.initialiseDevices();
        }

        if (isTimer) {
            SessionLauncher.initializeTimer();
        }
        for (int i = 0; i < NUMBER_ITERATIONS; i++) {
            System.out.println("Iteration number " + (i+1));
            this.appTester = new AppTester();
            this.appTester.testAllApp();
        }
}
```

Listing 4.3: SESSIONLAUNCHER Code snippet for starting a testing session

Dataset and single app cycles. APPTESTER and CMDEXECUTOR represent the core components of the whole testing process. Indeed, APPTESTER can be viewed as brain of the process, since it tells step-by-step to the body, *i.e.*, the CMDEXECUTOR component, which commands it has to execute and at what stage of the process it has to perform it.
Listing 4.4 shows a very simplified code snippet of the relation between the two above mentioned components.

First of all, APPTESTER creates a for-loop in which it iterates each APK file contained in the APKs directory. Once again, this directory is specified in the CONFIGURATIONMANAGER. The first if statement checks whether the file in question has an adequate extension, *i.e.*, it is able to be installed on a android mobile device. After that, APPTESTER uninstalls the concerned APK, so that at each iteration of the testing it gets reinstalled. This because, it may be that an APK gets affected by some previously generated sequences (*e.g.*, a sequence of random events that led the app to an external website).
Afterwards, APPTESTER checks which automated tool has been chosen by the tester, so that it can tell to the CMDEXECUTOR component, which command-line it has to execute. As stated before, APPTESTER prepares the single testing components such as which APK, which tool, which testing parameters, etc., while CMDEXECUTOR executes them without any prior knowledge.
Once the automated tool has been detected, CMDEXECUTOR is able to concretely start the *single app cycle*, executing the passed command-line. This is represented in Listing 4.4 by the method *generateReport()* (*line 33*). Indeed, CMDEXECUTOR uses a single instance of the Java-class *Runtime* that allows the application to interact with the environment in which the app is running [43]. This is actually achieved by the *Runtime.getRuntime()* (*line 34*). The next line concretely executes the passed command-line.

```java
/**
* @class: AppTester
*/
public void testAllApp() {
        for (File apk : this.apksDirectory) {
            if (apk.getName().endsWith(".apk") && !apk.isDirectory()) {
                    uninstallApp(apk.getName());
                    installApp(apk.getName());
                    if (IS_MONKEY) {
                        testAppWithMonkey(config.getMonkeyRepDir(), apk.getName
    ());
                    } else if (IS_SAPIENZ) {
                        testAppWithSapienz(config.getSapienzRepDir(), apk.
    getName());
                    }
            }
        }
        // waiting to threads to finish
        File testLog = CmdExecutor.getCurrentLog();
        if (hasCrash(testLog)) {
            generateCrashLog(testLog);
        }
}
private void testAppWithSapienz(String dest, final String APK_NAME) {
  CmdExecutor.generateReport(dest, CommandLines.SAPIENZ_CMD_LINE(APK_NAME));
}

private void testAppWithMonkey(String dest, final String APK_NAME) {
  CmdExecutor.generateReport(dest, CommandLines.MONKEY_CMD_LINE(APK_NAME));
}

/**
* @class: CmdExectutor
*/
public static void generateReport(String dest, String cmd){
        Runtime runtime = Runtime.getRuntime();
        Process p = runtime.exec(cmd);
        StreamGobbler output = new StreamGobbler(p.getInputStream(), cmd, dest)
    ;
        output.start();
        writeTestingEndTime(dest);
    }

public static File getCurrentLog() {
    return lastGeneratedLog();
}
```

Listing 4.4: Testing mechanism between AppTester and CmdExecutor

The returned code of the function call at *line 35* consists of a new *process* object. The process actually contains the execution of the command line. Assigning the execution of each single command-line to a new single separate process has advantages; In particular:

- Processes are independent from each other. If the execution of a command-line fails, it can be interrupted without affecting the entire testing process;
- Multithreading can be easily supported; Indeed, the component STREAMGOBBLER extends the *Thread* Java-class which implements the *Runnable* Java-interface. Each time a new process comes in, it starts a new thread in this class.
- Each process has it own timeout. It may be that some command-lines cannot properly terminate and need to be interrupted during their execution.

Reporting phase. STREAMGOBBLER, in turn, is in charge of writing the test report. Each time its constructor gets instantiated in the *generateReport()* method in the CMDEXECUTOR class, it starts a new thread and begins in parallel the writing phase of the log. As shown in Listing 4.5, the method *run()* overrides the method located in the *Runnable* interface. Indeed, it gets automatically invoked when in the *generateReport()* method an object of type *StreamGobbler* calls the function *start()* (*line 37*). Once it gets called, the *reporting* phase begins. This phase uses a *PrintWriter* as well as classic *Reader* for writing text on a file. Before the test log is written, the metadata about the testing environments are appended to the writer. At the end of the process the writer is closed and the thread can terminate. Once the thread is finished, the method *writeTestingEndTime()* (*line 38*) can be executed. It complements the metadata, writing the testing end time, so that the total testing time can be computed.

```java
/**
* @class: StreamGobbler
*/
@Override
public void run() {
        try {
            Writer writer = new PrintWriter(outputPath, "UTF-8");
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            String line;
            writer.append(TesterData.getMetaData()); // insert metadata
            while ((line = br.readLine()) != null) {
                System.out.println(" > " + line); // console overview
                writer.append(line).append("\n"); // test log
            }
            closeWriter();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
}
```

Listing 4.5: STREAMGOBBLER code snippet writing a test log

Listing 4.6 shows a short version of a test log of the app *com.danvelazsco.fbwrapper* that has been generated after the execution of MONKEY .

```
/**
 * Meta-data
 */
Tester Name: Lucas Pelloni
Testing Start Time: 05/04/2017 11:18:29
Testing End Time: 05/04/2017 11:48:30
Total Testing Time: 30 minutes (0.5 hours)
Type of testing: testing on a physical device
Device name: c0808bf731ab321
Percentage of motion events: 2.0% (number of motion events: 60 of 3000 events)
Percentage of system events: 6.0% (number of system events: 180 of 3000 events)
Percentage of touch events: 1.0% (number of touch events: 30 of 3000 events)

/**
 * Test log
 **/
:Monkey: seed=1495075565065 count=3000
:AllowPackage: com.danvelazco.fbwrapper
:IncludeCategory: android.intent.category.LAUNCHER
:IncludeCategory: android.intent.category.MONKEY
// Event percentages:
//   0: 1.0%
//   1: 2.0%
//   2: 2.4931507%
//   3: 18.698631%
//   4: -0.0%
//   5: 31.164383%
//   6: 18.698631%
//   7: 6.0%
//   8: 2.4931507%
//   9: 1.2465754%
//   10: 16.205479%
:Switch: #Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;end
    // Allowing start of Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER]
:Sending Trackball (ACTION_MOVE): 0:(4.0,4.0)
:Sending Trackball (ACTION_MOVE): 0:(4.0,-3.0)
    //[calendar_time:2017-05-05 09:48:23.894  system_uptime:717348]
    // Sending event #100
...
```

Listing 4.6: Test log of com.danvelazco.fbwrapper

As shown in the figure above, the test logs do not only contain the test results of the tested app, but also the above mentioned meta-data for documenting and retracing the whole testing session.

Investigating phase. The *single app cycle* in the "strict sense", *i.e.*, the stage where the APK gets stressed with an automated tool is over. At this point, the logs must be investigated about the possibility that some apps have generated a crash during its testing time frame. In this sense, the last part of the method *testAllApp()* illustrated in Listing 4.4, is in charge of stating whether a test log contains a crash or not. The method for checking whether a test log has collected a crash inside it is quite intuitive. This because, the syntax used by MONKEY and SAPIENZ in their report for indicating the presence of a crash is the same. As illustrated in Listing 4.7, a crash can be delimited using the following two *Strings*:

- Crash beginning: `"// CRASH: "`
- Crash end: `"// "`

```
...
:Sending Trackball (ACTION_MOVE): 0:(-4.0,-3.0)
:Sending Trackball (ACTION_MOVE): 0:(3.0,-1.0)
// CRASH: com.danvelazco.fbwrapper (pid 4302)
// Short Msg: java.lang.NullPointerException
// Long Msg: java.lang.NullPointerException
// Build Label: samsung/espressowifixx/espressowifi:4.2.2/JDQ39/P3110XXDMH1:user/release-keys
// Build Changelist: 8291
// Build Time: 1419156873000
// java.lang.NullPointerException
//   at com.danvelazco.fbwrapper.activity.BaseFacebookWebViewActivity.onKeyDown(BaseFacebookWebViewActivity.java:649)
//   at com.danvelazco.fbwrapper.FbWrapper.onKeyDown(FbWrapper.java:429)
//   at android.view.KeyEvent.dispatch(KeyEvent.java:2640)
//   at android.app.Activity.dispatchKeyEvent(Activity.java:2433)
//   at com.android.internal.policy.impl.PhoneWindow$DecorView.dispatchKeyEvent(PhoneWindow.java:2021)
//   at android.view.ViewRootImpl$ViewPostImeInputStage.processKeyEvent(ViewRootImpl.java:3845)
//   at android.view.ViewRootImpl$ViewPostImeInputStage.onProcess(ViewRootImpl.java:3819)
//   at android.view.ViewRootImpl$InputStage.deliver(ViewRootImpl.java:3392)
//   at android.view.ViewRootImpl$InputStage.onDeliverToNext(ViewRootImpl.java:3442)
//     ...
//
:Sending Touch (ACTION_DOWN): 0:(215.0,683.0)
:Sending Touch (ACTION_UP): 0:(163.15541,597.4464)
...
```

Listing 4.7: Crash log of com.danvelazco.fbwrapper illustrated within its test log

Indeed, the method *generateCrashLog()* is in charge of extracting the crash(es) from its test log. The parsing technique used by this method is to individuate the beginning of the crash using the `"START_CRASH"` String. Once the start has been individuated, the loop continues to insert lines of the log into a local array of Strings (*line 13*) until it finds the `"END_CRASH"` String. Once the end has been reached the loop terminates and CMDEXECUTOR writes the results into an external file, in order that the crash can be extracted.

```
1  /**
2  * @class: AppTester
3  */
4  public void generateCrashLog(File testLog) {
5          ...
6          ArrayList<String> crashLog = new ArrayList<>();
7          Pattern start = Pattern.compile(START_CRASH);
8          String line;
9          while ((line = in.readLine()) != null) {
10             Matcher matcher = start.matcher(line);
11             if (matcher.find()) {  // crash start
12                 while (!line.contains(END_CRASH)) { // crash end
13                     crashLog.add(line);
14                     line = in.readLine();
15                 }
16             }
17         }
18         CmdExecutor.writeToFile(crashLog, dest);
19     }
```

Listing 4.8: APPTESTER's method for extracting a crash log from its test log

At this point, the APK has been tested, reported, if they occurred, the relative crashes extracted. BECLOMA can now repeat this entire process with the next application (*i.e.*, the next iteration in the loop inside the method *testAllApp())*. This scenario is represented by the "yes" answer of the question *"are there still apps in the dataset which have not been tested in this dataset cycle?"* in the figure 3.3. Once all the applications specified in the dataset have been tested exactly one time, SESSIONLAUNCHER can begin with the next iteration in its loop (Listing 4.3) and so start testing the whole dataset another time. This scenario, in turn, is described by the "no" answer of the question above. Indeed, diagram 3.3 consequently starts the *dataset cycle* again. This loop ends when the number of iterations reaches the one specified by the user.
In addition, during each test iteration and at the end of the whole testing session useful statistics are computed and written into external excel files, using the components ONGOINGCALCULATOR and FINALCALCULATOR. They use the pure Java library *Apache POI*, for reading and writing files in Microsoft Office formats [44].

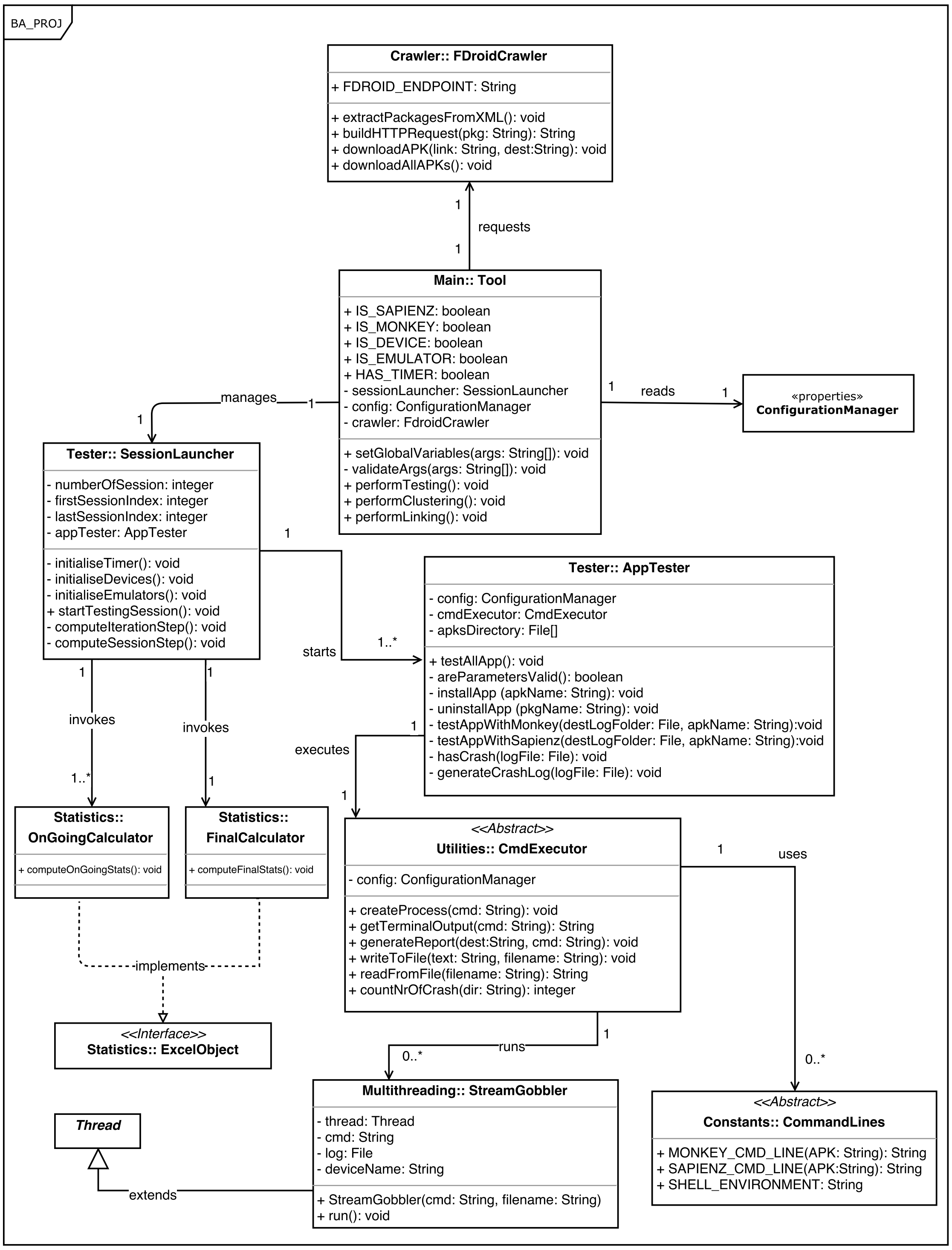
BA_PROJ
Crawler:: FDroidCrawler
+ FDROID_ENDPOINT: String
+ extractPackagesFromXML(): void
+ buildHTTPRequest(pkg: String): String
+ downloadAPK(link: String, dest:String): void
+ downloadAllAPKs(): void
1
requests
1
Main:: Tool
+ IS_SAPIENZ: boolean
+ IS_MONKEY: boolean
+ IS_DEVICE: boolean
+ IS_EMULATOR: boolean
+ HAS_TIMER: boolean
- sessionLauncher: SessionLauncher
- config: ConfigurationManager
- crawler: FdroidCrawler
+ setGlobalVariables(args: String[]): void
- validateArgs(args: String[]): void
+ performTesting(): void
+ performClustering(): void
+ performLinking(): void
1
reads
1
«properties»
ConfigurationManager
manages
1
1
Tester:: SessionLauncher
- numberOfSession: integer
- firstSessionIndex: integer
- lastSessionIndex: integer
- appTester: AppTester
- initialiseTimer(): void
- initialiseDevices(): void
- initialiseEmulators(): void
+ startTestingSession(): void
- computeIterationStep(): void
- computeSessionStep(): void
1
starts
1..*
Tester:: AppTester
- config: ConfigurationManager
- cmdExecutor: CmdExecutor
- apksDirectory: File[]
+ testAllApp(): void
- areParametersValid(): boolean
- installApp (apkName: String): void
- uninstallApp (pkgName: String): void
- testAppWithMonkey(destLogFolder: File, apkName: String):void
- testAppWithSapienz(destLogFolder: File, apkName: String):void
- hasCrash(logFile: File): void
- generateCrashLog(logFile: File): void
1
executes
1
1
invokes
1..*
1
invokes
1
Statistics:: OnGoingCalculator
+ computeOnGoingStats(): void
Statistics:: FinalCalculator
+ computeFinalStats(): void
implements
<<Interface>>
Statistics:: ExcelObject
<<Abstract>>
Utilities:: CmdExecutor
- config: ConfigurationManager
+ createProcess(cmd: String): void
+ getTerminalOutput(cmd: String): String
+ generateReport(dest:String, cmd: String): void
+ writeToFile(text: String, filename: String): void
+ readFromFile(filename: String): String
+ countNrOfCrash(dir: String): integer
1
uses
0..*
1
runs
0..*
Multithreading:: StreamGobbler
- thread: Thread
- cmd: String
- log: File
- deviceName: String
+ StreamGobbler(cmd: String, filename: String)
+ run(): void
Thread
extends
<<Abstract>>
Constants:: CommandLines
+ MONKEY_CMD_LINE(APK: String): String
+ SAPIENZ_CMD_LINE(APK:String): String
+ SHELL_ENVIRONMENT: String

Figure 4.1: Class diagram of the testing part of the tool

4.2 Clustering

First of all, BECLOMA individuates the directory in which all the previously generated crash logs have been stored. This directory is indicated again in the CONFIGURATIONMANAGER.
As shown in Listing 4.9, BECLOMA loops all the crashes contained inside it and for each of them it calls the method *extractCrashLog()*, in the CRASHLOGEXTRACTOR component.

```java
/**
* @class: Main
*/
CrashLogExtractor extractor = new CrashLogExtractor();
final String CRASH_CONTAINER = ConfigurationManager.getCrashContainerDir();
File[] crash_container = new File(CRASH_CONTAINER).listFiles();
for (File crash : crash_container) {
  extractor.extractCrashLog(crash);
}
```

Listing 4.9: CRASHLOGEXTRACTOR code snippet converting crash files into CrashLog objects

The method *extractCrashLog()* converts simple crash log files stored inside a folder into CRASHLOG Java-objects. Indeed, for each crash log file found inside the directory, the constructor of the CRASHLOG class get instantiated and so a new object of this type gets created. Each time the constructor of the CRASHLOG class gets invoked, it automatically creates the structure of the CRASHLOG-object analogously to the structure of the real crash log file.
For instance, the crash log described in Listing 3.1 is converted into a CRASHLOG-object represented in Listing 4.10. It is printed out using the trivial Java method *toString()*, which per default gives a String representation of the object in question.

```
Crash {
  crash_path = /Users/Lucas/Desktop/UZH/BA/CrashLogCollector/crash_log_com.ringdroid.txt
  packageName = com.ringdroid
  Short = android.database.StaleDataException
  Long = android.database.StaleDataException: Attempted to access a cursor after it has been closed.
  first_java_trace_line = android.database.StaleDataException: Attempted to access a cursor after it has been closed.
  trigger_method = [android.database.BulkCursorToCursorAdaptor.getCount(BulkCursorToCursorAdaptor.java:70)]
  trigger_class = BulkCursorToCursorAdaptor
  log_lines = [// CRASH: com.ringdroid (pid 20442), // Short Msg: android.database.StaleDataException, ...]
}
```

Listing 4.10: CRASHLOG-object

From Listing above we can observe how the CRASHLOG-object has assumed the same structure as the crash log file. Indeed, it analogously describes the *package name* to which the crash belongs, the *short* and the *long* explanations about the exception, the *trigger method*, etc. In addition to them, the CRASHLOG-object stores a list of Strings containing the log words, where each position of the array is occupied by a different line.

Preprocessing. In according to the approach explained in Section 3.2, the words of the crash reports must be preprocessed (*tokenized*) in order to prepare them to be compared to each other. In this sense, the CRASHLOG class statically invokes the method *tokenizeLine()* in the LUCENETOKENIZER class for each line contained in the crash log. Listing 4.11 shows how it concretely works.

```java
/**
* @class: CrashLog
*/
for (String line : this.logLines) {
  LuceneTokenizer.tokenizeLine(line);
}
```

Listing 4.11: Each line inside the crash report is tokenized using LUCENETOKENIZER

To provide a concrete example, the following line, extracted from the crash report in Listing 3.1 is passed as argument to the method *tokenizeLine()*:

```
//    at android.database.BulkCursorToCursorAdaptor.throwIfCursorIsClosed(BulkCursorToCursorAdaptor.java:64)
```

The following two boxes show the various tokenization steps that are performed by BECLOMA.

1. **Lucene StandardTokenizer**
   `at` `android` `database` `BulkCursorToCursorAdaptor` `throwIfCursorIsClosed` `BulkCursorToCursorAdaptor` `java` `64`
2. **BECLOMA CamelCase Tokenizer**
   `at` `android` `database` `Bulk` `Cursor` `To` `Cursor` `Adaptor` `throw` `If` `Cursor` `Is` `Closed` `Bulk` `Cursor` `To` `Cursor` `Adaptor` `java` `64`
3. **BECLOMA LowerCase Tokenizer**
   `at` `android` `database` `bulk` `cursor` `to` `cursor` `adaptor` `throw` `if` `cursor` `is` `closed` `bulk` `cursor` `to` `cursor` `adaptor` `java` `64`

As shown above, the crash log line gets firstly preprocessed using the *StandardTokenizer* provided by Lucene. Afterwards, BECLOMA for the reasons explained in Section 3.2 performs a further tokenization step, tokenizing the lines using an additional *CamelCase tokenizer*. Finally, it uses a simply tokenizer to make the words lower case. This tokenization process is repeated for all lines of all crash reports stored in the directory. At the end of this process, each CRASHLOG-object has its own set of preprocessed log lines (in class diagram 4.4, represented by the list of Strings *crashLogWords*)

TF-IDF In order to follow the clustering approach described in Section 3.2, BECLOMA creates for each crash log its correspondent vector space model. For this purpose, it makes use of the **Lucene** library again. Indeed, the approach proposed by Lucene consists of (i) executing a indexation process with a set of Lucene documents, (ii) creating a term vector for each document (iii) computing for each term within the vector its tf-idf score.
Diagram 4.2 illustrates the main components of the Lucene indexation process.

Creating documents and adding fields. As a first step, BECLOMA generates a set of *Lucene documents*. To do this, it goes through all previously created CRASHLOG-objects and converts their preprocessed words into Lucene documents. Each document must contain its group of *fields* in order to be indexed. Therefore, when a new Lucene document is created, a set of fields must be set and enclosed inside it. BECLOMA indexes documents which enclose *text fields*, since they are already stored, tokenized and indexed. An in-depth explanation about fields and their types can be found on [14, 16]. In order to be able to compute tf-idf values for the indexed Lucene

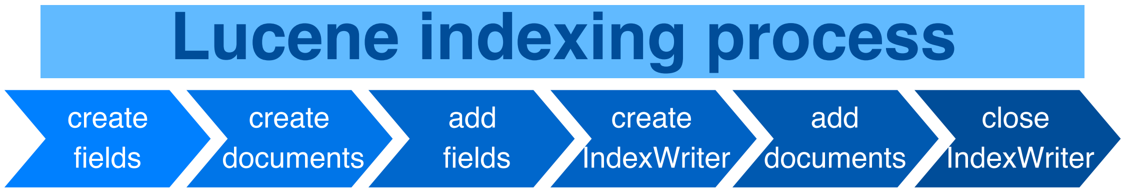

Figure 4.2: Apache Lucene indexing process

documents, BECLOMA must enable the property that they can have *term vectors* [18]. The first steps of the Lucene indexing process can be summarized by the following code snippet, which is a simplified version of the *createIndex()* method, illustrated in diagram 4.4.

```
1  /**
2  * @class: TFIDFCalculator
3  */
4  private void createIndex(List<CrashLog> crashLogs) throws IOException {
5        // TODO: create IndexWriter
6              for (CrashLog crash : crashLogs) {
7                  ArrayList<String> crash_log_words = crash.getSetOfWords();
8                  // type of field is set
9                  FieldType fieldType = new FieldType(TextField.TYPE_STORED);
10                 // term vector is enabled
11                 fieldType.setStoreTermVectors(true);
12                 // new document is created
13                 Document doc = new Document();
14                 for (String word : crash_log_words) {
15                     // fields are added into the documents
16                     doc.add(new Field(LConstants.FIELD_NAME, word, fieldType));
17                 }
18                 doc.add(new Field(LConstants.FIELD_ID, crash.getPath(),
       fieldType));
19             }
```

Listing 4.12: TFIDFCALCULATOR describing the Lucene indexing process

As shown in Listing above, BECLOMA goes through all the given crash logs and stores in a local list their preprocessed log words. Afterwards, the field type *TextField* is selected (*line 9*). In the next line, the above mentioned term vector property is enabled. Then, a new Lucene document is generated. At this point, BECLOMA goes through each preprocessed word and at each iteration adds the current word to a new field which has just been created (*line 16*). The new field has a name, a content, *i.e.*, the word in question and a type, *i.e.*, the previously selected field type. Finally, the entire field is added to the document. At the end of the loop an additional field is added and used as ID for the document (in our case, the crash log path acts as unique attribute in the index).

Creating IndexWriter and adding documents. In order to index Lucene documents, BECLOMA must generate an *IndexWriter* [?]. To achieve this, it creates an object of that type. However, the instantiating process of this class requires some supplementary configurations that must be passed

to the constructor in order to create a new object of that type. These two pieces of information consist in (i) the directory where the index should point at and (ii) the Lucene *Analyser* which is in charge of analysing the indexed documents. After the specification of these two information, a new object of type *IndexWriter* can be created. Once created, all documents can be added to the index. At the end of the indexing process the writer must be closed.
Listing below complements the code snippet 4.12 with the instantiation of the *IndexWriter* class and consequentially with the addition of the documents to the index.

```
 1 /**
 2 * @class: TFIDFCalculator
 3 */
 4 private void createIndex (List<CrashLog> crashLogs) throws IOException {
 5       // directory gets specified
 6         FSDirectory dir = FSDirectory.open(new File("\Users\Lucas\Desktop\BA\
    Index").toPath());
 7         // configuration is set
 8             IndexWriterConfig config = new IndexWriterConfig(new
    StandardAnalyzer());
 9             // IndexWriter gets instantiated
10             IndexWriter writer = new IndexWriter(dir, config);
11
12             for (CrashLog crash : crashLogs) {
13                 ArrayList<String> crash_log_words = crash.getSetOfWords();
14                 FieldType fieldType = new FieldType(TextField.TYPE_STORED);
15                 fieldType.setStoreTermVectors(true);
16                 Document doc = new Document();
17                 for (String word : crash_log_words) {
18                     doc.add(new Field(LConstants.FIELD_NAME, word, fieldType));
19                 }
20                 doc.add(new Field(LConstants.FIELD_ID, crash.getPath(),
    fieldType));
21                 // document are added to the index using an IndexWriter object
22                 writer.addDocument(doc);
23         }
24                 writer.close();
25 }
```

Listing 4.13: TFIDFCALCULATOR describing the instantiation of an IndexWriter

As illustrated in the example above, the location where the index should point at can be inserted using the Lucene class *FSDirectory* [15] (*line 6*). In the next line of code, the analyser which is used for analysing the Lucene documents is defined. This can be made using a *IndexWriterConfig* [**?**], which holds all the configuration that is used to create an IndexWriter (*line 8*). BECLOMA defines again a *Standard Analyser* [**?**], the same used for preprocessing the crash logs. Finally, a new *IndexWriter* can be constructed per the previously configured settings. Once an *IndexWriter* is created, all Lucene documents can be added to the index using a simple method called *addDocument()*. At the end of the indexing process all preprocessed words contained inside the CRASHLOG-objects have been converted into Lucene documents and have been indexed.

Computing TF-IDF scores. At this point, the index has been created and each document has the possibility of storing its vector of terms. Thus, tf-idf values can now be computed for each

Table 4.1: Structure of the hash map containing its term vector

Crash log path (key)	*HashMap (value)*	
../Desktop/BA/CrashLogCollector/crash_log1_com.ringdroid.txt	*term (key)*	*tfidf (value)*
	exception	1.73
	view	5.12
	accessibility	4.77
	crash	1.00
	scrollable	1.92
	adapter	2.44
	ringdroid	1.00
	...	...

Table 4.2: Vector terms A and B before normalization

Vector A	
terms	**tf-idf**
handler	3.15
accessibility	1.7
crash	1.13
invoke	1.41
ringdroid	1.00

Vector B	
terms	**tf-idf**
exception	1.73
handler	2.01
accessibility	4.77
crash	1.00
scrollable	1.92
adapter	2.44
ringdroid	1.00

word of each vector of each crash report. In this direction, BECLOMA invokes the *computeScoreMap()* method in the TFIDFCALCULATOR class, which returns a hash map object. This hash map has as keys the unique locations of the crash logs, each one of them is associated with another hash map which contains the correspondent term vector. Table 4.1 shows the structure of the hash map, taking as a reference the crash log presented in Listing 3.1.

As shown in Table 4.1, the terms which have the highest tf-idf relevance are *accessibility* and *view*. Indeed, they have a high term frequency in the currently scored crash log and a low document frequency in the entire collection. Terms such as *crash* or *ringdroid* have a low tf-idf weight. In fact, they are generic and irrelevant terms, since they don't give any useful information about the topic of the document.

Cosine similarity. In order to state whether two crash logs refer to the same bug, *i.e.*, they can belong to the same group in the bucket, BECLOMA follows the approach described in Section 3.2. Indeed, it computes a well-known similarity measure, *i.e.*, cosine similarity [46] between different crash reports. However, as a necessary preliminary step, the two vectors referring to the two different crash logs have first be *normalized*. The *normalization* process described in the approach (Section 3.2) is reported in the example shown in Tables 4.2 and 4.3. As illustrated, the vectors A and B after the normalization process have the same length and have been complemented with their missing words.

Computing Cosine Similarity to create the bucket. Once the vector terms have been normalized, the cosine similarity among crash reports can be computed. In this direction, BECLOMA

Table 4.3: Normalized weighted vector terms A and B

Vector A		*Vector B*	
terms	**tf-idf**	**terms**	**tf-idf**
exception	**0**	exception	1.73
handler	3.15	handler	2.01
accessibility	1.7	accessibility	4.77
crash	1.13	crash	1.00
scrollable	**0**	scrollable	1.92
adapter	**0**	adapter	2.44
invoke	1.41	**invoke**	**0**
ringdroid	1.00	ringdroid	1.00

defines a threshold in the class ORACLE which represents the tolerance for evaluating the similarity between two crash reports. Indeed, if the calculated cosine similarity among them is greater than the given limit, the two crash logs are considered to describe the same bug, *i.e.*, they will belong to the same bug group in the bucket.

Concretely, BECLOMA invokes the *fillCrashLogBucket()* method, which is in charge of comparing the crash logs, classifying them with the aim of filling the bucket. The bucket consists of a hash map, which has as keys a set of Strings which act as a labels for the entire bug group they represent. Each label, in turn, has its own list of CRASHLOG-objects which have been considered by the ORACLE referring to the same bug.
The bucketing process iterates all crash reports and it concretely works as follows:

1. The method *fillCrashLogBucket()* firstly extracts the first crash log in the list and insert it into the bucket, assigning to it the label of the first bug group.

2. Starting from the second iteration, BECLOMA computes the cosine similarity between the current crash log and all crash logs which are placed at the first position in the list of the other bug groups.

3. Whether all computed similarities are smaller than the threshold provided by the ORACLE means that there is no crash logs in the bucket already which can be considered the same as the current one. For this reason, BECLOMA creates a new bug group with a new label and insert into it the current crash log.
   Otherwise, in the event that there is at least one computed similarity which is greater than the threshold, BECLOMA extracts the group inside the bucket which shows the highest similarity with the crash log and insert it into it. In doing so, it gets added into the group which "resemble it more".

Diagram 4.3 shows and summarizes the above mentioned bucketing process provided by BECLOMA.

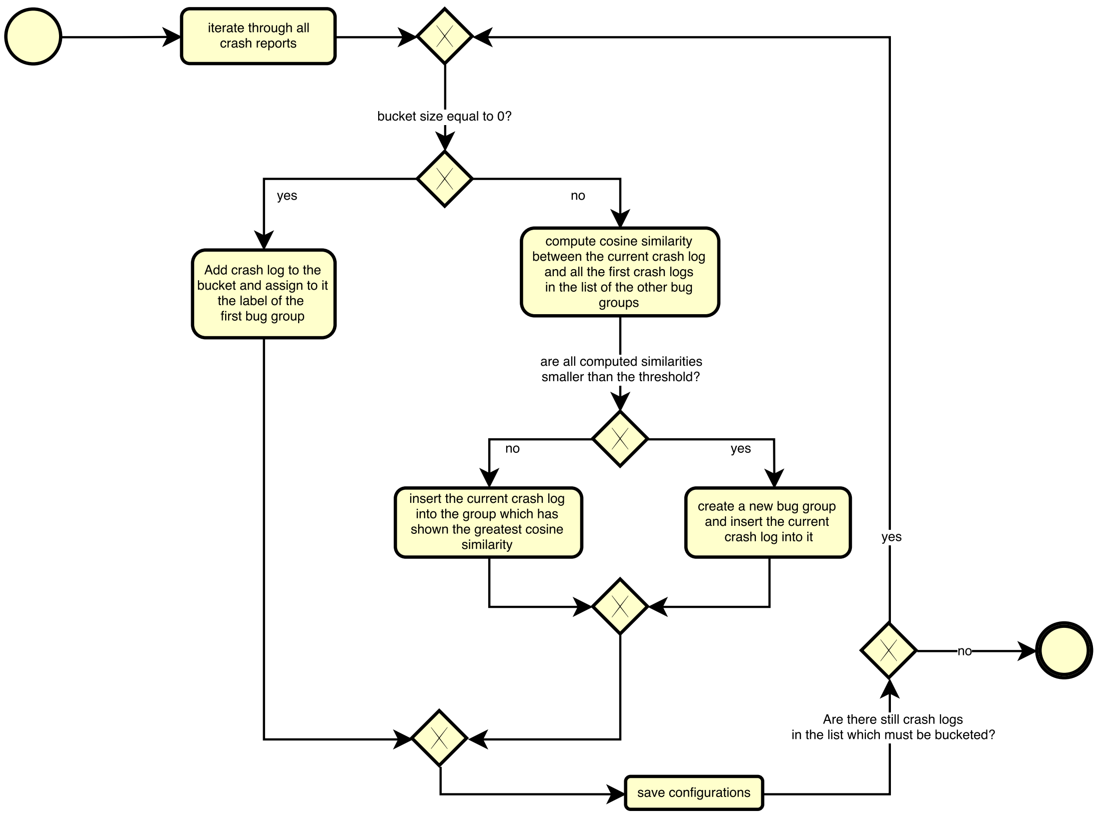
iterate through all crash reports
bucket size equal to 0?
yes
no
Add crash log to the bucket and assign to it the label of the first bug group
compute cosine similarity between the current crash log and all the first crash logs in the list of the other bug groups
are all computed similarities smaller than the threshold?
no
yes
insert the current crash log into the group which has shown the greatest cosine similarity
create a new bug group and insert the current crash log into it
yes
no
Are there still crash logs in the list which must be bucketed?
save configurations

Figure 4.3: BPMN diagram describing the bucketing process

At the end of the clustering phase, all crash logs collected in the testing phase have been systematically classified inside the bucket.

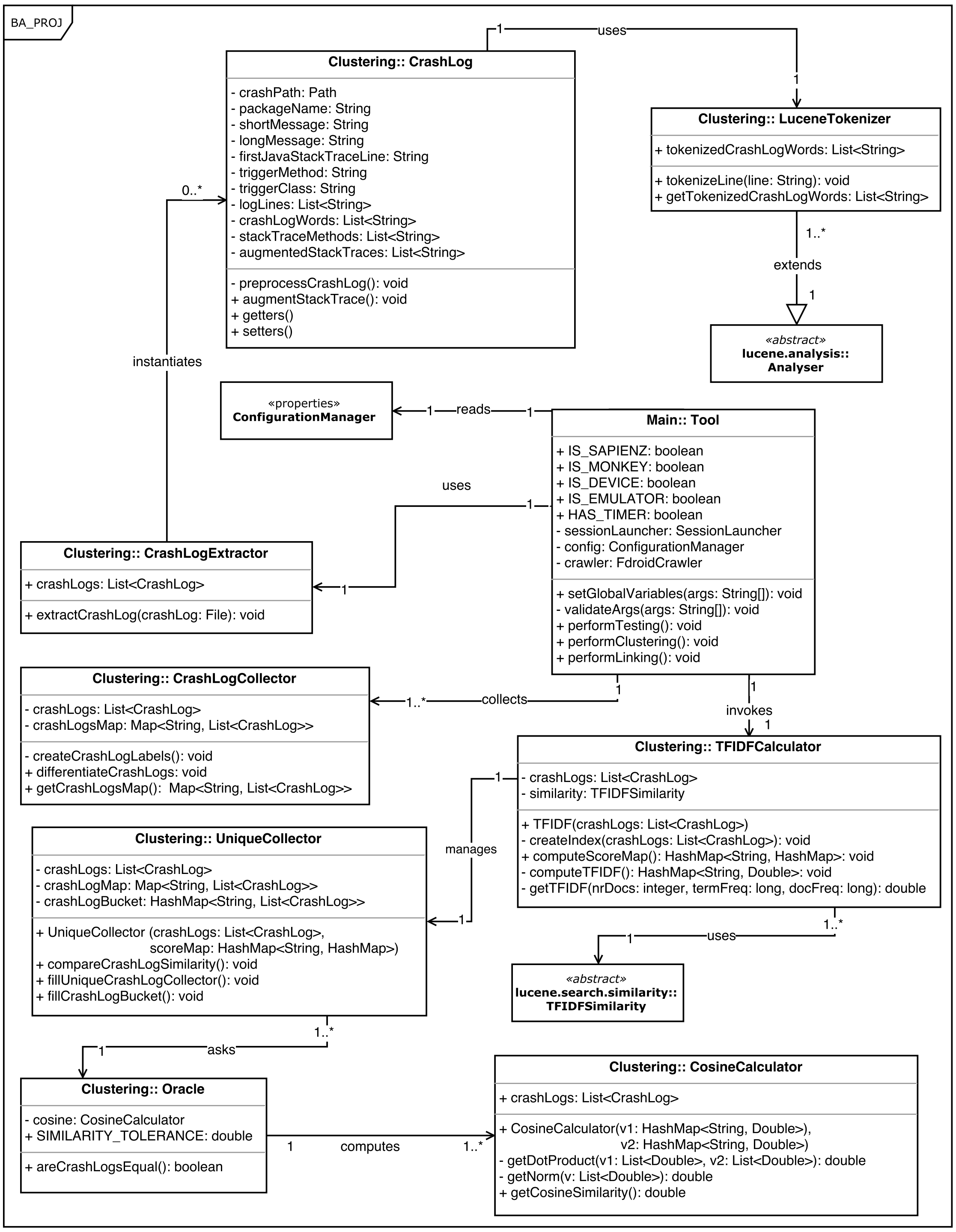
BA_PROJ
Clustering:: CrashLog
- crashPath: Path
- packageName: String
- shortMessage: String
- longMessage: String
- firstJavaStackTraceLine: String
- triggerMethod: String
- triggerClass: String
- logLines: List<String>
- crashLogWords: List<String>
- stackTraceMethods: List<String>
- augmentedStackTraces: List<String>
- preprocessCrashLog(): void
+ augmentStackTrace(): void
+ getters()
+ setters()
uses
Clustering:: LuceneTokenizer
+ tokenizedCrashLogWords: List<String>
+ tokenizeLine(line: String): void
+ getTokenizedCrashLogWords: List<String>
extends
«abstract» lucene.analysis:: Analyser
instantiates
«properties» ConfigurationManager
reads
Main:: Tool
+ IS_SAPIENZ: boolean
+ IS_MONKEY: boolean
+ IS_DEVICE: boolean
+ IS_EMULATOR: boolean
+ HAS_TIMER: boolean
- sessionLauncher: SessionLauncher
- config: ConfigurationManager
- crawler: FdroidCrawler
+ setGlobalVariables(args: String[]): void
- validateArgs(args: String[]): void
+ performTesting(): void
+ performClustering(): void
+ performLinking(): void
Clustering:: CrashLogExtractor
+ crashLogs: List<CrashLog>
+ extractCrashLog(crashLog: File): void
Clustering:: CrashLogCollector
- crashLogs: List<CrashLog>
- crashLogsMap: Map<String, List<CrashLog>>
- createCrashLogLabels(): void
+ differentiateCrashLogs: void
+ getCrashLogsMap(): Map<String, List<CrashLog>>
collects
invokes
Clustering:: TFIDFCalculator
- crashLogs: List<CrashLog>
- similarity: TFIDFSimilarity
+ TFIDF(crashLogs: List<CrashLog>)
- createIndex(crashLogs: List<CrashLog>): void
+ computeScoreMap(): HashMap<String, HashMap>: void
- computeTFIDF(): HashMap<String, Double>: void
- getTFIDF(nrDocs: integer, termFreq: long, docFreq: long): double
manages
Clustering:: UniqueCollector
- crashLogs: List<CrashLog>
- crashLogMap: Map<String, List<CrashLog>>
- crashLogBucket: HashMap<String, List<CrashLog>>
+ UniqueCollector (crashLogs: List<CrashLog>, scoreMap: HashMap<String, HashMap>)
+ compareCrashLogSimilarity(): void
+ fillUniqueCrashLogCollector(): void
+ fillCrashLogBucket(): void
«abstract» lucene.search.similarity:: TFIDFSimilarity
asks
Clustering:: Oracle
- cosine: CosineCalculator
+ SIMILARITY_TOLERANCE: double
+ areCrashLogsEqual(): boolean
computes
Clustering:: CosineCalculator
+ crashLogs: List<CrashLog>
+ CosineCalculator(v1: HashMap<String, Double>), v2: HashMap<String, Double>)
- getDotProduct(v1: List<Double>, v2: List<Double>): double
- getNorm(v: List<Double>): double
+ getCosineSimilarity(): double

Figure 4.4: Class diagram of the clustering part of the tool

Table 4.4: Scheme of the Reviewmap

Package name (key)	*Review list (value)*
com.danvelazco.fbwrapper	Review object 1
	Review object 2
	Review object 3
	...
com.amaze.filemanager	Review object 1
	Review object 2
	...

4.3 Linking

As mentioned in the introduction of Chapter 3, the linking phase performed by BECLOMA is in charge of studying the complementarity between user feedback and the outcomes of automated testing tools, *i.e.*, crash logs.

Extracting user reviews. The first step in the linking phase performed by BECLOMA is to read the dataset location, which contains the set of preprocessed and classified user reviews, specified in CONFIGURATIONMANAGER. Afterwards, BECLOMA uses the REVIEWCOLLECTOR component illustrated in diagram 4.5, in order to convert all user feedbacks located in the dataset to a set of Java-objects. To achieve this, each time BECLOMA finds a new review in the dataset, it instantiates the class REVIEW, creating a new object of that type. Afterwards, this object gets systematically inserted into a *ReviewMap*, located in the REVIEWCOLLECTOR class. This hash map has as keys a set of labels indicating the names of the packages for which the reviews have been submitted. Thus, each label is associated with its own list of REVIEW-objects. Table 4.4 depicts a shortened version of the architecture of the *ReviewMap*. At this point, BECLOMA has at disposal two different sources information, *i.e.*, the set of CRASHLOG-*objects* collected and classified inside the previously constructed bucket and a set of preprocessed and categorized *user reviews*.
Before the beginning of the linking process, the set of preprocessed words belonging to each CRASHLOG-object must be *augmented* in according to the approach explained in Section 3.3.
In this sense, BECLOMA firstly extracts each CRASHLOG-object from the bucket and then invokes for each of them the method *augmentstackTrace()* in the CRASHLOG class. Concretely, the augmenting process works as follows:

1. As explained in Section 3.3, for each CRASHLOG-object only the name and the cause of the raised exceptions are selected, *i.e.*, the String attribute *firstJavaStackTraceLine*. Taking again as reference the crash log illustrated in the figure 3.1, the name and cause of the exception can be found at *line 5*. After the preprocessing procedure has been completed, the line looks like the following:

   android, database, stale, attempted, access, cursor closed, bulk, adaptor, count .

   As you can see, the line has been already "stemmed and cleaned" using the method *stemmAndCleanReports()* in the AUGMENTER class. This method preprocesses the reports exploiting the Information Retrieval techniques explained at the beginning of Section 3.3. Indeed, programming keywords such as "*if*", "*throw*" or "*java*" and too short words such as "*to*" or "*at*" have been removed after the preprocessing procedure.

2. After cleaning the reports, the remaining text is augmented with the source code methods included in the stack trace. In this direction, BECLOMA firstly extracts all methods present

in the stack trace. To achieve this, it goes through all log lines in the method *augmentStack-Trace()* and for each line it compiles the regular expression positioned at *line 7* in Listing 4.14.

```
 1 /**
 2 * @class: CrashLog
 3 */
 4 public void augmentStackTrace()() {
 5         for (String line : this.logLines) {
 6             if (line.startsWith("//   at") && line.contains(".java:")) {
 7                 String[] temp = line.split("\\(")[0].split("\\.");
 8                 String method = temp[temp.length-1];
 9                 this.stackTraceMethods.add(method);
10             }
11         }
12         ...
13 }
```

Listing 4.14: Regular expression for extracting all methods from a stack trace

The first if statement checks whether the current line belongs to the stack trace body, thus starting from the line number number 4 in the example 3.1. At *line 7*, BECLOMA extracts only the method which is involved in the current line. For instance, the method which will be extracted from the following crash log line:

android.database.BulkCursorToCursorAdaptor.throwIfCursorIsClosed(BulkCursorToCursorAdaptor.java:64)

will be

throwIfCursorIsClosed

Indeed, the regular expression firstly splits the line by a *round bracket* creating a local static array of Strings. Afterwards, it selects the first position in this array. So, it splits the just created array by a *period*, forming a second array of Strings. This array is stored into the variable called *temp* (*line 7*). Finally, it selects the last position of this latter. The figure below depicts the splitting process for the crash log line represented above:

- `line.split("\\(")`
  android.database.BulkCursorToCursorAdaptor.throwIfCursorIsClose,
  BulkCursorToCursorAdaptor.java:64)
- `line.split("\\(")[0]`
  android.database.BulkCursorToCursorAdaptor.throwIfCursorIsClose
- `String[] temp = line.split("\\(")[0].split("\\.") // length = 4`
  android,
  database,
  BulkCursorToCursorAdaptor,
  throwIfCursorIsClose
- `String method = temp[temp.length-1] // temp[3]`
  throwIfCursorIsClose

Once all methods present in the trace stack have been extracted, BECLOMA controls whether in the list of methods coming from the source code there are elements in common with those coming from the stack traces. In the event that there are, all additional words, *i.e.*, a list of keywords concerning that method in the source code, are added to the above illustrated *firstJavaStackTraceLine*. Code snippet below summarizes the core of the *augmenting* process:

```
1  /**
2  * @class: Augmenter
3  */
4  public List<String> getAdditionalWordsFromSourceCode(){
5      HashMap<String, String> methods = Augmenter.getMethodsFromCSV();
6      List<String> augmentedStackTrace = convertToList(this.
       firstJavaStackTraceLine);
7      methods.forEach((pMethod, additionalWords)-> {
8        if (this.stackTraceMethods.contains(pMethod)) {
9          augmentedStackTrace.add(additionalWords);
10       }
11     });
12     return augmentedStackTrace;
13 }
```

Listing 4.15: Augmenting a stack trace with methods coming from the source code

First of all, the words coming from the source code are stored inside a local hash map called *methods*. This hash map has as keys a set of Strings, which represent the name of the methods found while parsing the source code. Their values consists of a set of Strings, each of them contains a list of additional words split by a white space. *Line 7* goes through each key element in the hash map and whether BECLOMA notices that there is an element in common between the list of stack trace methods and those from the source code (*line 8*), it adds the associated additional words to the variable *augmentedStackTrace* (*line 9*). At the end of this process, the *firstJavaStackTraceLine* has been converted into a variable called *augmentedStackTrace* and possibly augmented with the set of additional words for those methods in common. This variable now looks like the following:

android, database, stale, attempted, access, cursor closed, bulk, adaptor, count,
sound, hear, context, supported, filename, rate, channels, launch, intent, title, ...

At this point, BECLOMA has at disposal a set of *preprocessed* reviews and a set of *augmented* stack traces. The final step provides the linking procedure using the *asymmetric Dice* similarity coefficient described in Section 3.3.
First, BECLOMA reads the Dice threshold declared inside the Linker class, which represents the minimal score the *preprocessed reviews* and the *augmented stack traces* must obtain in order to be considered as linked. In the approach described in Section 3.3, pairs of documents having a Dice score higher than **0.5** were considered as linked by the approach. In order to compute that coefficient, BECLOMA passes as arguments in the constructor in the Linker class the pairs of documents, both in the form of a list of Strings. Afterwards, Linker invokes the method *evaluateLinking()*, which refers to the *Façade pattern*. Indeed, the SimilarityMetricsFacade component reproduces the design pattern *Facade*, which provides a simplified interface for evaluating the asymmetric Dice. Figure 4.16 shows how it concretely works.

```java
/**
* @class: SimilarityMetricsFacade
* @return a double value representing the similarity between the given texts
*/
public static double evaluateAsimmetricDiceIndex(List<String> pReview, List<String> augmentedLog) {
        return AsimmetricDiceIndex.computeAsimmetricDiceIndex(pReview, augmentedLog);
}
```

Listing 4.16: SIMILARITYMETRICSFACADE implementing the design pattern *Facade*

After defining how the Dice coefficient is computed, BECLOMA must just go through all CRASHLOG-objects, extracts for each of them its *augmentedStackTrace* variable mentioned above. Then, iterates the whole set of *preprocessed reviews* stored in the *ReviewMap* illustrated in Table 4.4, and computes for each review the Dice coefficient among it and the *augmentedStackTrace* variable. Finally, if the Dice score results greater than the threshold (in our case, 0.5), BECLOMA writes the correspondent *review ID* with its set of preprocessed words and the set of words representing the augmented stack trace of the current crash log into an external *csv*-File. To achieve this, it uses the component WRITER located in the package *Utilities*. At the end of the entire linking process, BECLOMA writes some metadata into the *csv*-File such as the date, the number of linked reviews, etc. The figure 4.17 shows a simplified code snippet representing the linking procedure described above.

```java
/**
* @class: Main
*/
public void performLinking() {
  crashLogBucket.forEach((crash_log_location, crashlogs) -> {
    for (CrashLog crashLog : crashlogs) {
       List<Review> reviewList = reviewMap.get(crashLog.getPackageName());
       for (Review review: reviewList) {
         List<String> preprocessedReview = review.getPreProcessedReview();
         List<String> augmentedStackTrace = crashLog.getAugmentedStackTraces();
         double diceIndex = SimilarityMetricsFacade.evaluateAsimmetricDiceIndex(preProcessedReview, augmentedStackTrace);
         if (diceIndex >= DICE_THRESHOLD) {
         String content +=  "review_id:   " + review.getId() + ", "
               + "\n pReview:   " + preProcessedReview.toString() + ", "
               + "\n aStackTrace:  " + augmentedStackTrace.toString() + ", "
               + "\n DiceScore:    " + diceIndex + "\n";
        }
      }
    }
  }
  Writer.writeOnFile(content, Paths.get("linking_results.csv");
}
```

Listing 4.17: Linking process

First, the list of crash logs stored for each bug group in the bucket gets iterated (*line 5-6*). Afterwards, the correct list of review is individuated by specifying the package name of the crash log (*line 7*). This is possible because the *reviewMap* has a set of keys the name of packages for which the reviews have been submitted. Then, each *Review* in the just created *reviewList* gets iterated in the *line 8*. At this point, two list of Strings store the preprocessed words of the current review as well the augmented stack trace of the current crash log (*line 9-10*. Later, using the SIMILARITYMETRICSFACADE component the Dice score among the pairs of documents is computed (*line 11*). *Line 12* checks whether the calculated coefficient is greater thatn the threshold. If this is the case, the *content* String (*line 13*) is updated with the *review id*, the set of words indicating the preprocessed review and finally the augmented stack trace. At the end of the entire linking procedure (*line 21*), the *content* String is written into an external file located at the specified *path*.

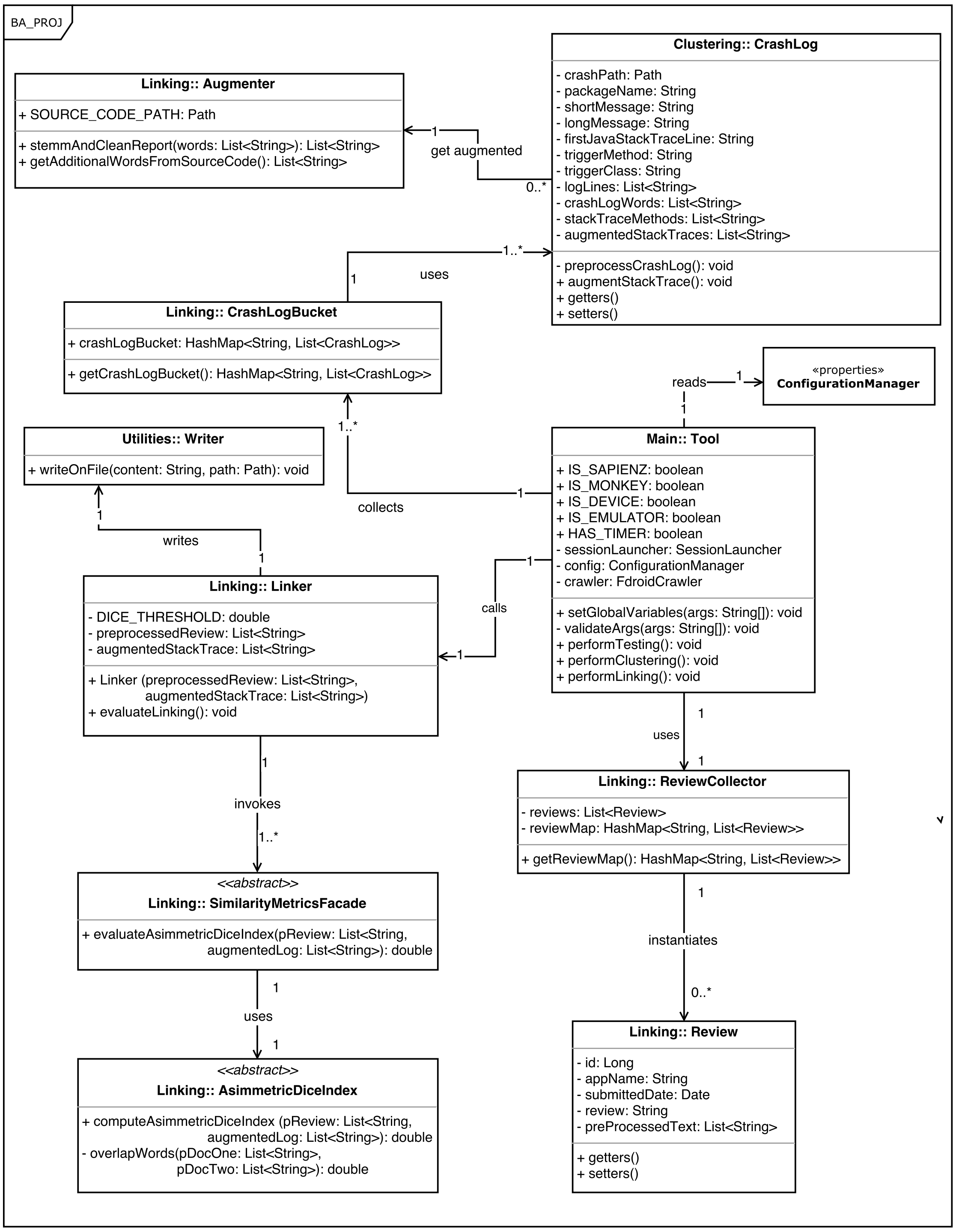
BA_PROJ
Clustering:: CrashLog
- crashPath: Path
- packageName: String
- shortMessage: String
- longMessage: String
- firstJavaStackTraceLine: String
- triggerMethod: String
- triggerClass: String
- logLines: List<String>
- crashLogWords: List<String>
- stackTraceMethods: List<String>
- augmentedStackTraces: List<String>
- preprocessCrashLog(): void
+ augmentStackTrace(): void
+ getters()
+ setters()
Linking:: Augmenter
+ SOURCE_CODE_PATH: Path
+ stemmAndCleanReport(words: List<String>): List<String>
+ getAdditionalWordsFromSourceCode(): List<String>
get augmented
1
0..*
uses
1
1..*
Linking:: CrashLogBucket
+ crashLogBucket: HashMap<String, List<CrashLog>>
+ getCrashLogBucket(): HashMap<String, List<CrashLog>>
«properties»
ConfigurationManager
reads
1
1
Utilities:: Writer
+ writeOnFile(content: String, path: Path): void
1..*
collects
1
Main:: Tool
+ IS_SAPIENZ: boolean
+ IS_MONKEY: boolean
+ IS_DEVICE: boolean
+ IS_EMULATOR: boolean
+ HAS_TIMER: boolean
- sessionLauncher: SessionLauncher
- config: ConfigurationManager
- crawler: FdroidCrawler
+ setGlobalVariables(args: String[]): void
- validateArgs(args: String[]): void
+ performTesting(): void
+ performClustering(): void
+ performLinking(): void
writes
1
1
calls
1
1
Linking:: Linker
- DICE_THRESHOLD: double
- preprocessedReview: List<String>
- augmentedStackTrace: List<String>
+ Linker (preprocessedReview: List<String>, augmentedStackTrace: List<String>)
+ evaluateLinking(): void
uses
1
1
Linking:: ReviewCollector
- reviews: List<Review>
- reviewMap: HashMap<String, List<Review>>
+ getReviewMap(): HashMap<String, List<Review>>
invokes
1
1..*
<<abstract>>
Linking:: SimilarityMetricsFacade
+ evaluateAsimmetricDiceIndex(pReview: List<String, augmentedLog: List<String>): double
instantiates
1
0..*
uses
1
1
<<abstract>>
Linking:: AsimmetricDiceIndex
+ computeAsimmetricDiceIndex (pReview: List<String, augmentedLog: List<String>): double
- overlapWords(pDocOne: List<String>, pDocTwo: List<String>): double
Linking:: Review
- id: Long
- appName: String
- submittedDate: Date
- review: String
- preProcessedText: List<String>
+ getters()
+ setters()

Figure 4.5: Class diagram of the linking part of the tool

4.4 Installation and usage of BECLOMA

BECLOMA is still in a experimental stage. Thus, the implementation of it has been tested just with *Android 4.4* and *Mac OS 10.11*. Augmenting the compatibility of it with multiple operating systems is plan of my future agenda.

4.4.1 General Configuration

First of all, in order to be able to use BECLOMA some external components must be installed and configured. The following shows the environment configurations that must be applied (they must be sequentially installed).

- **Brew**
  Brew [24] is a manager for installing the missing packages for Mac OS. It can be installed running the following command line on a Mac OS:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

- **Android SDK Platform-Tool**
  In order to easily install the Android SDK Platform-Tools component, which is required for launching SAPIENZ and MONKEY just download and install *Android Studio*, the official IDE for Android [4]. The download link can be found at:

```
https://developer.android.com/studio/index.html
```

- **pip**
  Pip is a manager for installing python packages on a Mac OS. It is needed for installing some python dependencies used by SAPIENZ. It can be easily installed running the following command line:

```
$brew install pip
```

or if it fails

```
$ sudo easy_install pip
```

4.4.2 Sapienz Environment Configuration

In order to be able to perform a testing session using SAPIENZ, the *emulators* respectively the attached *devices* must have installed the **API 19** level (version code *KitKat*) [3]. The following describes the configurations steps which have to be performed for using SAPIENZ.

- **Environment variables**
  Include the following *aapt* path as environment variable in the bash profile:

```
/Users/<user_name>/Library/Android/sdk/build-tools/19.x.x
```

- **Python dependencies**
  Install some required python *dependencies* using the *requirements.txt* file located in the SAPIENZ directory:

```
$ sudo pip install -r requirements.txt
```

- **Python version**
  The adequate python *version*(*2.7*) must be installed. Despite the correct version is installed, there may be a problem with the installation of the python dependencies. If that were the case, the following command line must be executed:

```
$ brew install python
```

4.4.3 Monkey Environment Configuration

The only requirement to use monkey is that the package of *Android SDK Platform-Tool* has been successfully installed.

4.4.4 Settings

In order to start a testing session a set of specifications must be declared in the CONFIGURATION-MANAGER file. This file configures the initial settings and the set of all parameters that will be processed by BECLOMA during the testing process. It is located in the BECLOMA source code and is renamed as *config.properties*. The most important properties have been already presented in the figure 4.1. In this sense, the following provides a detailed description about the testing parameters presented in the figure 4.1 [20, 31]:

- MONKEY
  - VERBOSITY ([*-v, -v-v-v*])
    It defines the verbosity of the log. Each additional *-v* increases the verbosity level.
  - RANDOM_EVENTS (*int*)
    Number of random events that will be generated by MONKEY in the testing cycle of a single APK.
  - DELAY_BETWEEN_EVENTS (*millisec*)
    It inserts a delay between the above specified number of random events.
  - PERCENTAGE_TOUCH_EVENTS (*int*, [0,100])
    It adjusts the percentage of *touch* events of the total number of events (*e.g.*, down-up events in the screen)
  - PERCENTAGE_SYSTEM_EVENTS (*int*, [0,100])
    It adjusts the percentage of *system* events of the total number of events (*e.g.*, volume controls)
  - PERCENTAGE_MOTION_EVENTS (*int*, [0,100])
    It adjusts the percentage of *motion* events of the total number of events (*e.g.*, random movements)
  - IGNORE_CRASHES (*boolean*, [true,false])
    It defines whether MONKEY stops when a crash or an unhandled exception occurs.

- SAPIENZ
  - SEQUENCE_LENGTH_MIN (*int*)
    Minimum number of events that will be generated for each SAPIENZ cycle.
  - SEQUENCE_LENGTH_MAX (*int*)
    Maximum number of events that will be generated for each SAPIENZ cycle.
  - SUITE_SIZE (*int*)
    The number of chromosomes (*i.e.*, test cases) that an individual (*i.e.*, test suite) owns.
  - POPULATION_SIZE (*int*)
    Numbers of individuals in the population in the genetic algorithm.
  - OFFSPRING_SIZE (*int*)
    The numbers of elements the offspring has in the whole test suite variation operator.
  - GENERATION (*int*)
    It sets the maximum generation number.
  - CXPB (*double*, [0,1])
    It defines the crossover probability used by the genetic algorithm.
  - MUTPB (*double*, [0,1])
    It defines the mutation probability used by the genetic algorithm.

In addition to them, the following specifications must be inserted:

- EMULATOR_NAME
  The name of the emulator on which the tests will be performed;
- AVD_BOOT_DELAY (*sec*)
  This time indicates how many seconds BECLOMA has to *wait*, when the attached devices or the emulators get rebooted;
- TIME_UNIT (*sec, min, hours*)
  It specifies the time unit recognized by the following *timeouts*;
- MONKEY_TIMEOUT
  It indicates after how long the testing cycle of a single APK tested with MONKEY gets automatically interrupted so that the testing of the next app can start. These time frames can be very useful since there may happen that some devices, after generating a very large number of *UI events* are not longer able to process new incoming events, freezing the entire Android system.
- SAPIENZ_TIMEOUT
  Same as above, only for SAPIENZ.
- ADB_EXEC_DIR
  It indicates the location of the executable file *abd* (Android Debug Bridge), a command-line tool which facilitates the communication between the user and the Android OS.

Table 4.5: Command-line arguments supported by BECLOMA

Supported arguments	**Description**
-device	Performs the testing on a physical attached android device
-emulator	Performs the testing on a virtual specified android emulator
-monkey	The dataset will be tested using monkey
-sapienz	The dataset will be tested using sapient
-timer	Optional. Starts a timer for a better overview during a testing session

4.4.5 BECLOMA usage

After specifying settings and parameters in the CONFIGURATIONMANAGER file, BECLOMA is ready to be started. The line below shows the usage of the BECLOMA command-line:

```
$ java -jar BECLOMA.jar [-emulator | -device] [-monkey | -sapienz] [timer]
```

Table 4.5 lists all of the supported BECLOMA arguments and explains their meaning.

Chapter 5

Results and Discussion

We evaluated the BECLOMA testing approach by conducting an empirical study on a smaller dataset of apps *i.e.*, 60 APKs. Afterwards, we used the collected crash logs for evaluating the correctness and reliability of our clustering approach, forming a bucket of unique crash reports. Finally, we chose the 3 apps that seemed most relevant to our study with enough reports generated by the testing tools and suitable reviews for investigating the accuracy of our linking approach.

5.1 Stack Trace Extraction

To conduct our experiment, we selected a dataset of 60 APKs, grouping them together exploiting the FDroid-crawler we built. We made our selection as follows: we tried to chose a dataset containing the most varied number of APKs. Indeed, we chose very popular apps which have a high number of downloads (*e.g., Telegram*) as well as less known apps with a low number of downloads (*e.g., Ringdroid*). The entire dataset can be found in the folder "Dataset" in the BECLOMA source code. Afterwards, we tested the whole dataset 3 times with the Android testing tools, i.e., MONKEY and SAPIENZ , running each tool for 30 minutes per app. According to the testing cycles described in Section 3.1, this can be translated as follows:

1. Number of iteration characterizing the *session cycle*: **3 iterations**;
2. Number of apps forming the *dataset cycle*: **60 apps**;
3. Time frame describing the *single app cycle*: **30 min**.

We conducted our experiment in the following environment: 2 Samsung Galaxy Tab 8 inches, with Android Kitkat 4.4 (API 19) on which we run the automated testing tools and Mac OS 10.11 for starting BECLOMA. The testing parameters we inserted in the settings file are summarized in Table 5.1.

The reason behind the parameters we chosen for MONKEY is based on experimental results. Indeed, we conducted few demo-experiments in order to state which combination of parameters led to the greater number of crashes. Furthermore, we changed the percentage of the system, motion and touch events at the beginning of each *dataset cycle* in order to variate our testing strategy. The selection behind the SAPIENZ parameters is consistent with the empirical study conducted by Mao *et al.* [31]. At the end of each session cycle we used our clustering approach to deduplicate the collected crash logs and form a bucket of unique ones.

Table 5.2 shows the results of the empirical study we conducted. As shown in Table, SAPIENZ revealed the largest number of unique crash logs in each iteration of the experiment. However, it should be noted that the number of total crashes revealed by MONKEY in the third dataset cycle is

Table 5.1: Chosen parameters for SAPIENZ and MONKEY to conduct our empirical study

Automated testing tool	**Chosen parameters**
Monkey	*verbosity = -v -v -v*
	random events = 3000
	delay between events = 10
	percentage touch events = 8
	percentage system events = 8
	percentage motion events = 8
	ignore crashes = true
Sapienz	*min sequence = 20*
	max sequence = 500
	suite size = 5
	population size = 50
	offspring size = 50
	generation = 100
	crossover = 0.7
	mutation = 0.3

Table 5.2: Testing results

Iteration number	**Crashes**	***Monkey***	***Sapienz***
First dataset cycle	App crashed	7	16
	Unique crashes	14	19
	Total crashes	39	45
Second dataset cycle	App crashed	12	23
	Unique crashes	21	28
	Total crashes	46	57
Third dataset cycle	App crashed	23	21
	Unique crashes	13	20
	Total crashes	87	54

clear greater that the ones found by SAPIENZ . This because, the percentage of *system* events has conspicuously grown (from 8% to 35%). This caused some sequences of events of that type that could not be elaborated by the devices, leading the app to failure. In fact, the number of unique crashes for MONKEY is not greater than the ones found by SAPIENZ , since they do not represent crashes caused by the application under test, but they can be categorized as native crashes.

5.2 Linking approach

The following Subsections present the results we obtained of our empirical study. Subsection 5.2.1 shows the precision of our linking approach, answering the $\textbf{RQ}_2$, while Subsection 5.2.2 presents our evaluation about the complementarity among stack traces and users feedback, answering $\textbf{RQ}_3$.

Table 5.3: Precision of the linking procedure

App	Precision
com.amaze.filemanager	60%
com.danvelazsco.fbwrapper	72%
com.ringdroid	64%
Average	**65%**

5.2.1 Precision Linking Approach

RQ$_2$ *To what extent can we link the defects arose from automated testing tools?*

To answer the **RQ**$_2$, we exploited the linking procedure provided by BECLOMA. Concretely, we selected from the previously tested dataset the 3 apps which seemed most relevant to our study with enough unique crash reports generated by MONKEY and SAPIENZ and a set of adequate reviews. The apps we chosen are *com.amaze.filemanager*, *com.danvelazsco.fbwrapper* and *com.ringdroid*. We limited our study to a small group of apps because the time constraints and the high effort to manually validate the data. Table 5.3 shows the precision values obtained by the linking approach implemented by BECLOMA over our limited dataset. The results reported in this Table confirm the accuracy and reliability of our linking procedure. Indeed, it shows its effectiveness in achieving quite high precision scores, *i.e.*, the percentage of correctly retrieved links was **65%**. We can affirm, that our approach performs in an accurate manner, providing a precise correlation between stack traces and crash-related user feedbacks.

5.2.2 Complementarity of Stack Traces and User Reviews

RQ$_3$ *How complementary are the two source of information? Can we leverage on both of them to increase the effectiveness of the testing process?*

To answer **RQ**$_3$, we firstly identified those links which correctly correlated a stack trace with its correspondent user review. Secondly, we categorized different type of issues concerning the crash-related user reviews and the stack traces, computing the followings metrics:

- I_C: % of issues reported in both reviews and crash logs;
- I_R: % of issues reported only in user reviews;
- I_T: % of issues reported only in crash logs.

In order to compute these metrics, we first defined T_{issues} as the total number of issues discovered for a given app. Therefore, consider L_{pos} as the number of unique true positive links between crash reports and crash-related user feedbacks revealed by BECLOMA. Afterwards, let be C_{logs} the number of crash reports which have been extracted but remain unlinked to any review. Then, consider R_{crash} as the number of user reviews, for which there exists no correlation to any crash report. Finally, we can define the relation between the three above proposed metrics as follows:

$$T_{issues} = L_{pos} + C_{logs} + R_{crash}$$

Thus, we can formally introduce the three overlap metrics introduced above as follow:

$$I_C = \frac{L_{pos}}{T_{issues}} \qquad I_R = \frac{R_{crash}}{T_{issues}} \qquad I_T = \frac{C_{logs}}{T_{issues}}$$

Table 5.4: Complementarity of Reviews and Stack Traces

App	$\mathbf{I_C}$	$\mathbf{I_R}$	$\mathbf{I_T}$
com.amaze.filemanager	17%	50%	33%
com.danvelazsco.fbwrapper	45%	45%	10%
com.ringdroid	24%	62%	14%
Average	**29%**	**52%**	**19%**

Table 5.4 shows the achieved results for our reduced dataset. As we can see from the it, the results concerning the I_R values (*i.e.*, the percentage of issues we can only find in user reviews) show in average greater scores than the I_T values (*i.e.*, the percentage of issue encountered uniquely through automated tools). Indeed, the number of issues reported only in user reviews is, on average, 52%. This percentage is conspicuously greater than the number expressing the other two metrics (in fact, I_C shows 29% and I_C only 19%).
From these results, we can draw an unambiguous conclusion, *i.e.*, that the number of issues highlighted by the stack traces (I_T), obtained through the execution of automated testing tools, is substantially *smaller* that the one identifying the number of issues reported in user reviews (I_C). Thus, we can summarize our first finding as follows:

$$I_T < I_C < I_R$$

As highlighted in Table 5.4, the number of common issues detected when relying on both user reviews and stack traces is, on average, only 29%. To find the cause of a such low percentage, we manually analysed the user reviews of our reduced dataset. Indeed, we found that the content of the reviews tend to describe the scenario of a crash which occurs during a particular input event. However, such scenario may imply some sensible sequences of events, which are hard to randomly replicate by automated testing tools.
For instance, for the `com.danvelazsco.fbwrapper` app, the review shown below claims about a crash occurring during a particular swipe input event, which is probably hardly replicable by automated testing tools.

"Quick fix for messages crash Slide in from the right go to preferences and use either desktop version or basic version..."

Similarly, for the `com.ringdroid` app an user claim that:

"Force closes when I search Maybe the problem is my large library but it truly is unusable..."

In this case, the user imputes the crash to his large library: obviously such particular conditions are difficult to be reproduced with an automated testing tools. From these considerations, we can draw our second conclusion. In this sense, we can conclude that issues concerning user reviews are conceptually different to issues detected by the stack traces. However, despite the number of issues revealed by user feedback being the greatest, there can exist a sort of complementary between issues detected by stack traces and users feedback. Indeed, this is highlighted in Table 5.4 that for app `com.danvelazsco.fbwrapper` shows an example of that complementarity.

With our findings we believe that we confirmed the importance and relevance to integrate user reviews in the validation process of a mobile application. Our results are quite promising, since they concretely showed the complementary between both source of information. However, there are still some limitations in the automated testing tools, since in many cases they are not able to reproduce some sensible scenarios describe in the user feedback. We argue that, we might augment the information in stack traces with the aim to link more crash-related user reviews with their correspondent crash reports.

Chapter 6

Conclusions and Future Work

In my thesis we investigated the importance of integrating user reviews in the validation process of mobile applications. For this purpose, we introduced an approach called BECLOMA which is able to (i) test a set of apps and extract possible crashes, (ii) form a bucket of unique crash reports and (iii) investigate the complementarity between crash-related user reviews and the bucketed stack traces. With the aim to evaluate the performance of BECLOMA, we performed an empirical study over a dataset containing different types of mobile applications. First, we downloaded our sample of APKs from the *FDroid API* and we launched an experiment on them. Afterwards, we reported its results, *i.e.*, the total number of crashes and unique crashes which occurred during the testing phase. Then, we created a bucket of unique crash logs. For this purpose, we first manually built an oracle according to the crash reports we collected. Afterwards, we adapted its threshold in order to reproduce that bucket. Finally, we started our linking procedure by linking the crash-related user reviews and the stack traces we bucketed. Again, we reported its results in Chapter 5.

We strongly believe, that our findings might convince other mobile developers to perform such an *user-oriented testing* by validating the reliability of their mobile applications. Indeed, despite BECLOMA still remains in an experimental stage, we argue that our results are quite promising. However, we are conscious that we tested our approach with a small dataset and thus we may adapt both the clustering and the linking threshold whether BECLOMA would be used with another, maybe greater set of apps. With a new set of apps, we believe that BECLOMA would require some manual effort for setting the correct threshold.

We are convinced that our findings and preliminary results lay the foundations for further research into the field of *integration of user feedback into the testing process*. At this stage, BECLOMA operates with a relative small number of user reviews and mobile applications. It would be nice to test the effectiveness of its approach with a very large amount of both sources of information.

There are different directions to investigate for future work. First, it could be possible to introduce a new tool which would be able to (i) *summarize* stack traces and user reviews linked together, supporting the activities performed by developers in their bug fixing sessions. (ii) create a *prioritisation scheme* for the generated failures taking into account the user reviews and finally (iii) *generate* specific *test cases* directly from user reviews.
Furthermore, we plan to improve the *augmenting process* of the stack traces implemented by BECLOMA. Indeed, we are convinced that a better selection of the words which augment the stack traces would improve the linking scores between crash-related user and these stack traces. For instance, we could consider not only the source code methods included in the stack traces but also the common classes among them. This would imply a greater set of words for the stack traces which may results in a better linking score.

Finally, we could implement a further feature for BECLOMA which is in charge of additionally filtering the reviews. Indeed, our approach considers also these reviews which present a positive connotation, *i.e.*, those that not refer to any crash but just express a positive opinion about the given app. For instance, we could try to conceive a filter function which analyses the reviews and discards, according to an external text file containing some stop words, those which express a position opinion about the app.

Bibliography

[1] What is monkey testing? types, advantages and disadvantages. http://istqbexamcertification.com/what-is-monkey-testing-advantages-and-disadvantages/.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. volume 32, pages 53–59, 2015.

[3] Android. Android 4.4 apis. https://developer.android.com/about/versions/android-4.4.html.

[4] Android. Android studio, the official ide for android. https://developer.android.com/studio/index.html.

[5] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., 1990.

[7] J. C. Campbell, E. A. Santos, and A. Hindle. The unreasonable effectiveness of traditional information retrieval in crash report deduplication. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 269–280. ACM, 2016.

[8] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang. Ar-miner: Mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 767–778, New York, NY, USA, 2014. ACM.

[9] W. Choi. Swifthand. https://github.com/wtchoi/SwiftHand.

[10] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 429–440, Washington, DC, USA, 2015. IEEE Computer Society.

[11] E. D. Corporation. Global development population and demographics study. Technical report, goo.gl/SKelVs, 2016.

[12] W. E. W. Dijkstra. Edsger w. dijkstra. https://en.wikiquote.org/wiki/Edsger_W._Dijkstra.

[13] A. S. Foundation. Apache lucene core. https://it.wikipedia.org/wiki/Lucene.

[14] A. S. Foundation. Class field. https://lucene.apache.org/core/6_0_1/core/org/apache/lucene/document/Field.html.

[15] A. S. Foundation. Class fsdirectory. https://lucene.apache.org/core/6_4_0/core/org/apache/lucene/store/FSDirectory.html.

[16] A. S. Foundation. Class textfield. https://lucene.apache.org/core/6_0_1/core/org/apache/lucene/document/TextField.html.

[17] A. S. Foundation. Class tfidfsimilarity. https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.

[18] A. S. Foundation. Field termvector. https://lucene.apache.org/core/5_4_0/core/org/apache/lucene/document/Field.TermVector.html.

[19] M. Glinz and T. Fritz. *Kapitel 8: Testen von Software*. University of Zurich, 2006-2013.

[20] Google. Android monkey. http://developer.android.com/tools/help/monkey.html.

[21] G. Grano. Implementation and comparison of novel techniques for automated search based test data generation. Master's thesis, University of Salerno, 2015.

[22] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: Msr for app stores. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 108–111, June 2012.

[23] M. Harman, Y. Jia, and Y. Zhang. Achievements, Open Problems and Challenges for Search Based Software Testing. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–12. IEEE, Apr. 2015.

[24] Homebrew. Homebrew, the missing package manager for macos. https://brew.sh.

[25] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA, 2011. ACM.

[26] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, Oct 2013.

[27] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, May 2015.

[28] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 111–122, Piscataway, NJ, USA, 2015. IEEE Press.

[29] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[30] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

[31] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[32] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.

[33] A. M. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE*, volume 3, page 260, 2003.

[34] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[35] H. Muccini, A. Di Francesco, and P. Esposito. Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 29–35. IEEE Press, 2012.

[36] M. Nagappan and E. Shihab. Future Trends in Software Engineering Research for Mobile Apps. *Saner'15*, 2015.

[37] M. D. Network. Testing process. `https://msdn.microsoft.com/en-us/library/ms978235.aspx`.

[38] D. Pagano and B. Brügge. User involvement in software evolution practice: A case studykhalid:2015:ieee. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 953–962, Piscataway, NJ, USA, 2013. IEEE Press.

[39] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 291–300, Sept 2015.

[40] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE 2017, page to appear. ACM, 2018.

[41] A. D. Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 499–510, 2016.

[42] Statista. Number of apps available in leading app stores as of march 2017. `https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/`, Mar. 2017.

[43] O. Team. Class runtime, oracle official documentation. `https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html`.

[44] Wikipedia. Apache poi. `https://en.wikipedia.org/wiki/Apache_POI`.

[45] Wikipedia. black-box testing. `https://en.wikipedia.org/wiki/Black-box_testing`.

[46] Wikipedia. Cosine similarity. `https://en.wikipedia.org/wiki/Cosine_similarity`.

[47] Wikipedia. tf-idf. `https://en.wikipedia.org/wiki/TfâĂŞidf`.

[48] Wikipedia. white-box testing. `https://en.wikipedia.org/wiki/White-box_testing`.