Bachelor Thesis

June 25, 2017

GUI usability and testing of mobile applications

Example subtitle

Lucas Pelloni

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

supervised by

Prof. Dr. Harald C. Gall
Dr. Sebastiano Panichella
Giovanni Grano (PhD student)

University of Zurich[UZH]

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

Bachelor Thesis

GUI usability and testing of mobile applications

Example subtitle

Lucas Pelloni

University of Zurich[UZH]

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

Bachelor Thesis

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

URL: https://github.com/lucaspelloni2/BA_PROJ

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 was undoubtedly one of the most rewarding academic experiences of my life until now. During these six months, I learned to know more in-depth the programming world, appreciating more and more its challenges and its beauties. However, the realization of my thesis would not have been possible without the help of few 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 bachelor's 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 its guidance and support, for its 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 could "bang my head" against the new challenges and this 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 so in detail before. I thank also my two computer science schoolmates, Ile and Erion, who helped me a lot with their knowledge about 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 also some words for those people, who support me during the whole bachelor degree.

Abstract

Zusammenfassung

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The following sections in this chapter give an overview about the context in which the thesis is placed, the motivation behind it and a brief introduction of the research questions prepared for it.

Afterwards, chapter 2 surveys the literature in the field of automated mobile testing, presenting the main approaches in this field and their exploration strategies. After that, it presents the literature available about the usage of users reviews in software maintenance activities.

Chapter 3 gives a general overview about the approaches and methodologies available in the literature, implemented in the presented tool.

Chapter 4 presents TACL, a tool which firstly tests and reports a set of android mobile apps using either MONKEY or SAPIENZ. Secondly, creates a cluster with the in the previous testing phase generated crash logs. Finally, uses a linking procedure for combing a set of user reviews with the stack trace that have been grouped together in the previous step. The tool enables developers to link natural language contained inside user feedbacks and pieces of source code, such as names of classes, methods, functions, etc.

Chapter 5 describes the obtained results after an experiment conducted on a set of mobile apps (available on *FDroid* [1]) with their correspondent reviews.

Finally, chapter 6 closes the main body of the thesis with concluding comments and proposals for future work.

1.1 Context

Testing is the action of inspecting the behaviour of a program, with the intention of finding anomalies or errors [31]. The goal behind software testing is to reach the highest *test coverage* finding the largest number of *errors* 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 when they are executed.

Nowadays, software testing is widely recognised as an essential part of any software development process, presenting however an extremely expensive activity. This because, trying to test all combinations of all possible input values for an application [16] requires a lot of workforce and it is almost always unthinkable to reach a testing-coverage of 100%. The reason is that, testing needs to be performed under time and budget constraints [18] and big and complex applications might have a very large number of potential test scenarios, many of which are really difficult to predict. In this sense, *Dijkstra* [9] 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. But then, why try to test a program even knowing that a full coverage (and

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

so a complete validation of the software) cannot be reached? Well, the answer is quite simple. A program, that would be carefully tested and all bugs found would be consequently fixed, increase the probability that this software would behave as expected in the untested cases [16].

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 (*see later*). *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. You can imagine system testing as the last checkpoint before the end customer. Indeed, *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 drived by two testing methodologies [38, 44]:

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

With *black-box* testing, also called functional testing, the tester doesn't need to have any prior knowledge of the interior structure of the application. 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 will interact with the system's user interface by providing inputs and examining outputs without knowing how and where the inputs are worked upon. On the other side, with *white-box* testing, also *glass testing* or *open box testing* [18], the test cases are extrapolated from the internal software's structure. Indeed, the tester writes the test cases defining a paths through the code, which has to provide a sensible output.
In practice, the whole testing process is strictly structured and it gets implemented through the use of different testing models. Usually, the selection of one model rather than another has a huge impact on the final testing result is carried out. The most popular and applied testing process models in practice are the *V-model* [42], the *Waterfall model* [43] and the *Spiral model* [40].
As shown, there exist few testing steps, methodologies and different models to accompany a testing process of a traditional software. However, when a tester is intent to define and choose the right pieces for testing a software in a careful manner, the common goal behind the testing stays always the same, i.e. generate a *test suite* (a set of *test cases*), with the smallest number of test cases causing the largest number of failures in the system [18], in order to increase the reliability of the system. However as mentioned before, the process of finding the correct test scenarios, afterwards execute them and finally report their results and compare 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 [3]. For this reason, it is necessary to reduce them, trying to improve the effectiveness of the whole testing process, with the aim to automate it.

1.2 Motivation

With the advent of the *mobile age*, the traditional software market has been always more complemented with a new point of view. 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, [29], 149 billions downloaded only in 2016 [35] and 12 million mobile app developers (expected to reach up 14 million in 2020) maintaining them [8].
This majestic revolution in the IT sector has had also a huge impact on the software testing area. This because, mobile applications differ from traditional software and so differ also their 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 [20]. Furthermore, they are context-aware [29]. Therefore, each mobile application is aware of the environment in which it is running and provides inputs according to its current context. This has some implications for the testing, because a test case running on a specific context may lead to its expected results, while the same test case running on another environment may be error-prone. In fact, bugs related to contextual inputs are quite frequent [29].
For this reason, mobile applications require new specialized and different testing techniques [29] in order to ensure their reliability. In this sense, an extremely valid approach has been the GUI 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 [20, 24, 26, 29].
These techniques consist of automatically create 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 [29]. 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 [30]. One reason is that an automated event-test-generation tool is 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"* [24], etc.). This because, current solutions generate redundant/random inputs that are insufficient to properly simulate the human behaviour, thus leaving feature and crash bugs undetected until they are encountered by the users. Furthermore, the crash reports that they generate usually lack of contextual information and are difficult to understand and analyze [5, 21]. For these reasons, sometimes a time-consuming manual approach can be needed for testing an application [30].

However, GUI testing could not be the only approach to help developers find bugs in a mobile application. Nowadays, the exponential growth of the mobile stores offers an enormous amount of informations and feedbacks from users. In fact, these users feedback are playing more and more a paramount role in the development and maintenance of mobile applications. Therefore, another different strategy for giving further support to developers during the testing phase of their applications is to incorporate opinions and reviews of the end-users during the software's evolution process.

Indeed, information contained within *user review*s can be exploited to overcome the limitations of state-of-the-art tools for automated testing of mobile apps in the following ways [**?**]:

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

1.3 Motivation Example

Following, I 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. One such example 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 complains about the app sharing function that shows a white screen when activated. A report generated by one of available automated testing tools, SAPIENZ [26], 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)
```

The aim behind my thesis is to investigate the correlation between these *user reviews* and the *crash logs* generated from the automated state-of-art tools described in the previous sections. I combine these two sources of information implementing a linking procedure to add contextual details to the logs generated by automated mobile testing tools. I argue that this approach can be fruitfully used to support developers in determining which bugs should be addressed first, prioritizing the arose failures. It is worth to notice that such prioritization is directly guided by the user feedback with the aim to maximize their satisfaction and consequentially, the app rating. I argue that this investigation might improve the automated testing process and might help other developers in validating their mobile applications, using TACL, the implemented tool described in the *Chapter 4*.

1.4 Research questions

In this thesis, I conduct a preliminary study to investigate the first and most important point from the previous list: *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, I conducted my thesis to answer the following research questions:

- $\mathbf{RQ}_1$: *Which are the most important tools for automatic testing of Android applications?*
- $\mathbf{RQ}_2$: *To what extent can we link the defects arose from automated testing tools with user reviews?*
- $\mathbf{RQ}_3$: *What kind of defects are detected by state-of-art tools proposed for GUI testing? How these defects are effectively perceived by users?*

For answering the $\mathbf{RQ}_1$, I want to investigate which are the most important automated testing tools used for testing Android mobile applications. For this purpose, I present in the Chapter 2 the main research in this field, giving an accurate overview about the testing tools in general, their pros and cons and their exploration strategies.

For the $\mathbf{RQ}_2$, I implement a linking procedure which uses a experimental-based threshold in order to state whether a review refers to a specific *crash log*. Based on the result of this linking approach I am able to discuss and answer the second research question.

For the $\mathbf{RQ}_3$, TODO!

Subsubsection

Figure 1.1: imgs/seal logo

1.4.1 Subsection

Paragraph. Always with a point.

```
/**
 * Javadoc comment
 */
public class Foo {
   // line comment
   public void bar(int number) {
      if (number < 0) {
         return; /* block comment */
      }
   }
}
```

Listing 1.1: An example code snippet

Chapter 2

Related Work

In the following two sections, I 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.* [27].

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 [7,24]: *random testing* [17,24], *systematic testing* [25] and *model-based testing* [2,6,23].

Fuzz testing

When test automation does occur, it typically relies on Google's Android MONKEY command-line [17]. Since it comes directly integrated in Android Studio, the standard IDE for Android Development, it is regarded as the current state-of-practice [25]. This tool simply generates, for the specified Android applications, pseudo-random streams of user events into the system, with the goal to stress the *AUT*[1] [17]. The effort required for using MONKEY is very low [7]. 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 [17].
The kind of testing implemented by MONKEY follows a black-box approach. Despite the robustness, the user friendliness [7, 24] and the capacity to find out new bugs outside the stated scenarios [1], this tool may be inefficient if the *AUT* would require some human intelligence (*e.g.* a login field) for providing sensible inputs [24].
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, due also to the randomness in the testing strategy implemented by it [1].

Dynodroid [24] is also a random-based testing approach. However, this tool has been discovered being more efficient than MONKEY in the exploration process [7].
One of the reasons behind a better efficacy has been that DYNODROID is able to generate both *UI inputs* and *system events* (unlike MONKEY , which can only generate UI events) [7].
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 [24]. All these

[1] Application Under Test

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 [24]. 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 [7,24] and finally *executes* it. After the execution of that event the AUT reaches a new state and the cycle again 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 [24].

Systematic testing

The tools using a systematic explorations strategy rely on more sophisticated techniques, such as symbolic execution and evolutionary algorithms [7].

SAPIENZ [26] introduced a new Pareto multi-objective search-based technique (the first one to 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, whether the source code is available. Otherwise the "instrumenting-process" follows a *black-box* approach whether 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 [26]. 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 communicate 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 compute 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 [26] show that SAPIENZ is an out-performer in the automated mobile testing area. The table below compares the above mentioned tools MONKEY and DYNODROID with SAPIENZ , illustrating the strength of the SAPIENZ 's approach.

Table 2.1: Statistics on found crashes from the SAPIENZ experiment published on [26]

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

As represented in the table above, 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 [26]. Most of these tools [2, 6, 23, 28] generate UI events from models, which are either manually designed or created from XML configuration files [26].
For example, *SwiftHand* [6] uses a machine learning algorithm to learn a model of the current *AUT*. This final state machine model [7] 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 [6]. *SwiftHand*, in a similar way to MONKEY generates only touching and scrolling UI events and is not able to generate System events [7].

2.2 Usage of users reviews in Software maintenance activities

The concept of app store mining was first introduced by *Harman et al.* [19]. In this context, many researchers focused on the analysis of user reviews to support the maintenance and evolution of mobile applications [27]. Pagano *et al.* [32] empirically analyzed the feedback content and their impact on the user community, while Khalid *et al.* [22] conducted a study on iOS apps discovering 12 types of user complaints posted in reviews. Several approaches have been proposed with the aim to classify useful reviews. AR-MINER [5] represents the first approach proposed in the literature able to discern informative reviews. In this direction, in a recent work Panichella *et al.* introduced a tool called SURF (Summarizer of User Reviews Feedback), that is able to analyse the useful informations contained in app reviews and to performs a systematic summarisation of thousands of user reviews through the generation of an interactive agenda of recommended software changes [34].
Finally, Palomba *et al.* [33] proposed CHANGEADVISOR, a tool able to suggest the source code artifacts to maintain according to user feedback.

2.3 Choice and motivation for the used automated testing tools

For the implementation of TACL, 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 [25] and therefore is well supported and documented by the community, (ii) the effort for using it is very low [7] and (iii) being it 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 [26]) an out-performer 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

Chapter 4

TACL

The tool I implemented, called TACL, is a *java-based* tool aimed at helping further developers during the testing process of their Android mobile applications.
For giving a cleaner and more understandable explanation of how TACL works, I would like to split its key features into three main categories (which should also be executed in a sequential way, in order to exploit the whole TACL's potentiality):

1. the TESTING part is in charge of testing a given set of *APKs*, 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 they together and create a crash log *bucket*;

3. the LINKING part represents the core feature of TACL. It pre-processes a set of given *user reviews* as well as the set of the previously created crash logs, in order to prepare and "clean" them for the linking procedure. Afterwards, it investigates whether it exists a correlation between the stack traces and the user feedbacks, with the aim to link, whether possible, the reviews with the crash logs.

4.1 Testing

First of all, if no set of *APKs* is available yet, TACL can be exploited for downloading the needed mobile applications from the *F-Droid API*[1]. In this direction, as shown in the picture 4.5, the component FDROIDCRAWLER, is first in charge of parsing a static structured file (*e.g.* a *csv*-file format), which contains a set of android packages names. The path of this file is given in the CONFIGURATION MANAGER, which contains a set of static properties that get elaborated by TACL. Second, FDROID CRAWLER 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 FDROID CRAWLER or can also be manually created. Now, the second step is to prepare and configure the testing environment. All the parameters needed for starting a testing session have to be specified in the CONFIGURATION MANAGER. Figure 4.1, shows an example of a simplified set of parameters which must be given a priori in order to launch a testing session.

[1] LINK API

```
/**
 * Testing session specifications
 */
MINUTES_PER_APP = 30
NR_OF_ITERATIONS = 5

/**
* Test logs directories
*/
MONKEY_DIR = Reports/MonkeyReports
SAPIENZ_DIR = Reports/SapienzReports

/**
 * Monkey parameters
 */
LOG_VERBOSITY = -v
PACKAGE_ALLOWED = -p
NR_INJECTED_EVENTS = 5000
DELAY_BETWEEN_EVENTS = 10
PERCENTAGE_TOUCH_EVENTS = 15
PERCENTAGE_SYSTEM_EVENTS = 15
PERCENTAGE_MOTION_EVENTS = 15
IGNORE_CRASH = True

/**
* Sapienz parameters
*/
SEQUENCE_LENGTH_MIN = 20
SEQUENCE_LENGTH_MAX = 500
SUITE_SIZE = 5
POPULATION_SIZE = 50
OFFSPRING_SIZE = 50
GENERATION = 100
CXPB = 0.7
MUTPB = 0.3
```

Listing 4.1: Properties which get elaborated during the testing sessions

The figure above represents a part of the CONFIGURATION MANAGER, where all the testing parameters for MONKEY and SAPIENZ are specified. An in-depth explanation about these parameters concering MONKEY and SAPIENZ can be found on [17], respectively on [26].
In addition to them, the directories on which the generated test logs are going to be stored must be given as well as the specifications about the testing session. The properties about the testing session consist of two values:

- 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.
- NR_OF_ITERATIONS, specifies how many times the whole dataset will be tested.

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

30 min p/a * 10 apps * 5 iterations = 1500 min. (25 hours).

Once the environment testing variables have 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 MAIN *args* (as mentioned before in the section 2.2, the tools which can be selected are 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. TACL 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 the *Android 4.4, KitKat*.
The listing below shows an example of a combination of possible parameters that could be given as main arguments.

```
$ java -\toolname.jar -device -monkey -timer
```

Listing 4.2: TACL command line

Once the configuration phase is terminated, TACL is able to start the testing process. As shown in the picture 4.5, it manages the component SESSIONLAUNCHER, which is in charge of translating the previously specified testing properties into "java readable code" and initializing the testing session. Concretely, after TACL invokes SESSIONLAUNCHER all the 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 finally starts the testing session. The Listing 4.3 gives a simplified code snippet about the beginning of the testing process.

```
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

First of all, the total number of iterations specified in the CONFIGURATIONMANAGER is read and stored into a constant of type int. After that, all the attached devices (or the chosen emulators) gets statically initialized. According to the boolean variable *isTimer*, a timer may also be started. Afterwards, a for-loop starts where at each iteration the method *testAllApp()* gets invoked. The idea behind this, is that at each iteration a new object of type APPTESTER is created, so that each created object represents one testing loop of the dataset. For this reason, as shown in the figure 4.5, 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.

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.

```
/**
* @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

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 get reinstalled. This beacause, it may be that an *APK* gets affected by 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.
After the automated tool has been detected, CMDEXECUTOR is able to concretely start the testing, executing the passed command-line. This is represented in listing 4.4 by the method *generateReport*. 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 [36]. This is actually achieved by the *Runtime.getRuntime()*. The next line executes with the previously created object the given command-line. Since this method returns a new Process object, the result of the execution is assigned to a separate process.
Assigning the execution of each single command-line to a new single separate process brings with it many advantages:

- Processes are independent of 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.

STREAMGOBBLER, in turn, is in charge of writing the test report. Each time its construct get instantiated in the *generateReport()* method of 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()* overridden from the *Runnable* interface gets automatically per-default invoked when in the *generateReport* method an object of type *StreamGobbler* calls the *start()* method. Once the *start()* method is called, the writing phase starts. 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()* in the method *generateReport()* can start. This method complement the metadata writing the testing end time, so that the total testing time can be computed.

```
/**
* @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)
:Sending Trackball (ACTION_MOVE): 0:(2.0,-1.0)
:Sending Trackball (ACTION_MOVE): 0:(-5.0,2.0)
:Sending Trackball (ACTION_MOVE): 0:(-5.0,3.0)
   //[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.

The testing phase 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 the 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 the listing 4.7, a crash can be delimited using the following two *Strings*:

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

```
...
:Sending Trackball (ACTION_MOVE): 0:(3.0,3.0)
: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)
:Sending Touch (ACTION_DOWN): 0:(243.0,812.0)
...
```

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

Indeed, the method *generateCrashLog()* (4.8 in the *testAllApp()* 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 add lines of the log 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 to extract the crash.

```
/**
* @class: AppTester
*/
public void generateCrashLog(File testLog) {
      ...
      ArrayList<String> crashLog = new ArrayList<>();
      Pattern start = Pattern.compile(START_CRASH);
      String line;
      while ((line = in.readLine()) != null) {
         Matcher matcher = start.matcher(line);
         if (matcher.find()) { // crash start
            while (!line.contains(END_CRASH)) { // crash end
               crashLog.add(line);
               line = in.readLine();
            }
         }
      }
      CmdExecutor.writeToFile(crashLog, dest);
   }
```

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

At this point, the *APK* has been tested, reported, its test logs have been investigated and possible crashes have been extracted. Figure 4.1 summarizes the four components which characterizes the testing process of one application.

Figure 4.1: Four test steps performed by TACL of an application

Now, TACL is able to start testing the next application (*i.e.*, the next iteration of the loop inside the method *testAllApp()*). Once all the applications specified in the dataset have been tested

exactly one time, SESSIONLAUNCHER can begin with the next iteration of its loop (listing 4.3) and so start testing the whole dataset another time. 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 [37].

4.2 Clustering

Once the testing phase is finished, all the generated crash logs are stored in a given directory. After this phase, the only way to differentiate them inside this directory is the name of the package for which these crashes occurred. However, among these crash logs there may be a lot of redundancy, since more of them may refer to the same bug. The aim behind the Clustering phase is to create a bucket of unique crash logs. This means, that each crash log has to be compared with the others of the same package and according to some metrics that will be explained below, they must be smartly group together. Figure 4.2 shows the idea behind the Clustering process.

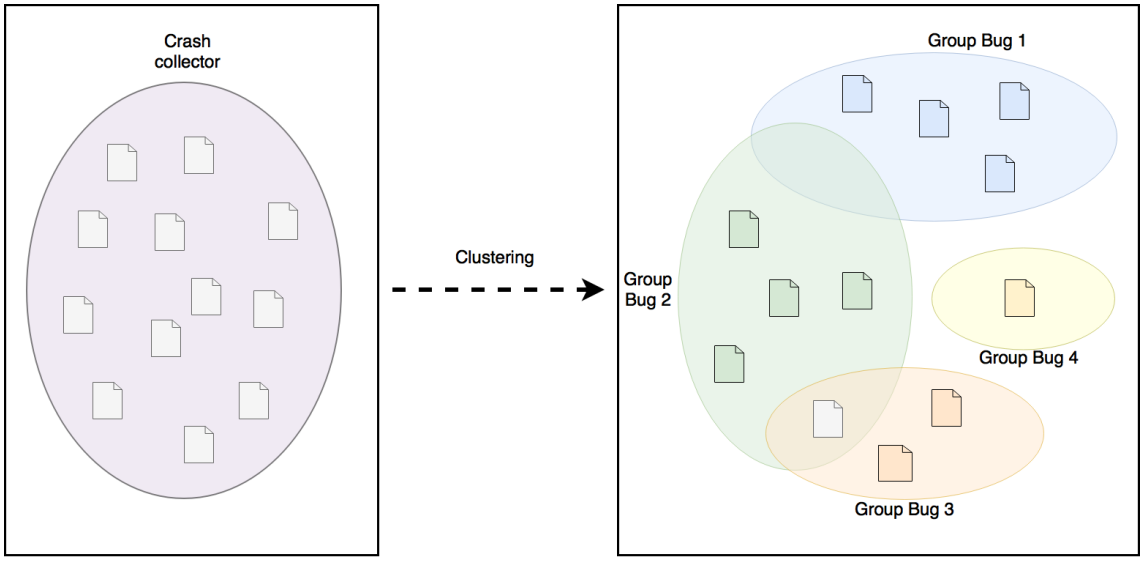

Figure 4.2: The idea behind the Clustering process

Some groups of crash logs may be overlapping. 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.

In order to understand better the Clustering approach, a clarification of how a crash log is structured must be done. In order to do this, another example of a crash log is given in the listing 4.9.

```
1. // CRASH: com.ringdroid (pid 6207)
2. // Short Msg: android.database.StaleDataException
3. // Long Msg: android.database.StaleDataException: Attempted to access a cursor after it has been closed.
```

```
4.  // android.database.StaleDataException: Attempted to access a cursor after it has been closed.
5.  //  at android.database.BulkCursorToCursorAdaptor.throwIfCursorIsClosed(BulkCursorToCursorAdaptor.java:64)
6.   //  at android.database.BulkCursorToCursorAdaptor.getCount(BulkCursorToCursorAdaptor.java:70)
7.  ...
```

Listing 4.9: 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. 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 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.

Before the explanation of the Clustering process can start, a premise must be made. All the classes that will be discussed below, refer to the class diagram represented in the figure 4.6.

First of all, TACL 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 the listing 4.10, TACL loops all the crashes contained inside it and for each of them it calls the method *extractCrashLog()*, in the CRASHLOGEXTRACTOR component.

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

Listing 4.10: 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 objects* analogously to the structure of the real crash log file.
For instance, the crash log described in the listing 4.9 is converted into the CRASHLOG-object represented in the listing 4.11. 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 4.11: CRASHLOG-object

As you can see in the listing above, 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. The second step of the clustering process is to *preprocess* the crash reports in order to prepare them to be compared to each other. To achieve this, all the words contained in the crash reports are preprocessed using **Apache Lucene** [10] well-known tokenization techniques. Indeed, the CRASHLOG class statically invokes for each line the method *tokenizeLine()* in the LUCENETOKENIZER class. Listing 4.12 shows how it concretely works.

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

Listing 4.12: Each line inside the crash report is tokenized using LUCENETOKENIZER

For the tokenization process, TACL uses a well known grammar-based Lucene-tokenizer, called *StandardTokenizer*. This tokenizer simply split the word fields into lexical units using punctuation and whitespaces as split points. In addition, it removes unnecessary symbols (*e.g.*, `"//"`). TACL converts all the strings to lowercase and extends the tokenizer with a further regular expression. This because, it's 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, TACL complements the StandardTokenizer of Lucene, so that CamelCase text fields can be also split at the upper-case letters into separate words.

Consider the following line in the crash report (listing 4.9) that 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 TACL.

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

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 is complemented with a new attribute which represents the set of preprocessed words.

TF-IDF. After the phase of preprocessing, the crash logs are ready to be bucketed. In order to classify them, some well-known bucketing techniques are implemented by TACL, with the aim to deduplicate the greatest number possible of crash reports. In this direction, TACL implements a *tf-idf* algorithm *(term frequency–inverse document frequency)*, a well-know term-weighting scheme used in information retrieval or text mining [41]. Indeed, tf–idf is a way to measure the relevance, the weight of a term compared to its document or its entire document collection (in our case, the crash logs). 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 [4].
Generally, a tf-idf algorithm consists in three main components [14]:

- **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 [41] describes a formula for calculating the term frequency:

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

  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, if a term is very common it returns a low value. The equation [41] 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.

TACL implements a *tf-idf* scheme, computing TF-IDF scores for all words within all crash logs using again the **Apache Lucene** library.
First of all, TACL creates an index using an `IndexWriter`. The following diagram illustrates the main components of the Lucene indexing process:

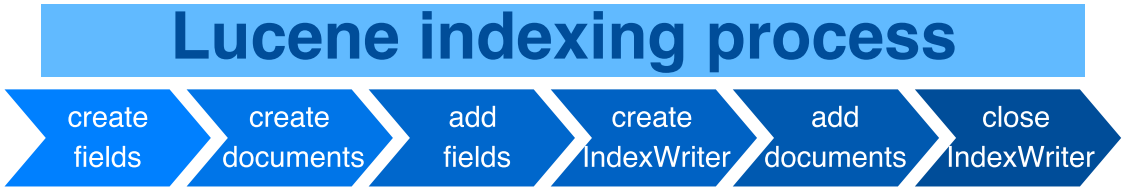

Figure 4.3: Apache Lucene indexing process

Creating documents and adding fields. The first step TACL performs is to create a set of *Lucene documents*. To do this, it goes through all previously created CRASHLOG-objects and convert their preprocessed words into Lucene documents. Each document must contain its group of *fields* in order to be indexed. Therefore, each time a new Lucene document is created, a set of fields must be set and enclosed inside it. A field can be viewed as a section of a document which can be optionally stored in the index [11] and usually has three components: name, type and content. For each stored field, it is important to determine which type is best suited to the content that is going to be indexed. For instance, TACL indexes documents which enclose *text fields*, since they are already stored, tokenized and indexed [13]. In order to be able to compute TF-IDF values for the indexed Lucene documents, TACL must enable the property that they can have *term vectors*, *i.e.*, they can store "a list of the document's terms and their number of occurrences in that document" [15].
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 the diagram 4.6.

```
/**
* @class: TFIDFCalculator
*/
private void createIndex(List<CrashLog> crashLogs) throws IOException {
        // TODO: create IndexWriter
        for (CrashLog crash : crashLogs) {
```

```
            ArrayList<String> crash_log_words = crash.getSetOfWords();
            // type of field is set
            FieldType fieldType = new FieldType(TextField.TYPE_STORED);
            // term vector is enabled
            fieldType.setStoreTermVectors(true);
            // new document is created
            Document doc = new Document();
            for (String word : crash_log_words) {
               // fields are added into the documents
               doc.add(new Field(LConstants.FIELD_NAME, word, fieldType));
            }
            doc.add(new Field(LConstants.FIELD_ID, crash.getPath(), fieldType));
         }
```

Listing 4.13: TFIDFCALCULATOR describing the Lucene indexing process

As shown in the listing above, TACL goes through all the given crash logs and stores in a local list their preprocessed log words. Afterwards, the field type *TextField* is selected. In the next line, the above mentioned term vector property is enabled. Then, a new Lucene document is generated. At this point, TACL goes through each preprocessed word and at each iteration adds the current word to a new field which has just been created. 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, TACL must generate an *IndexWriter*. This because, this object acts as a core component for creating and updating indexes. First of all, TACL creates an object of type *IndexWriter*. 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 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.
The listing below complements the code snippet 4.13 with the instantiation of the *IndexWriter* class and consequentially with the addition of the documents to the index.

```
/**
* @class: TFIDFCalculator
*/
private void createIndex (List<CrashLog> crashLogs) throws IOException {
         // directory gets specified
         FSDirectory dir = FSDirectory.open(new File("\Users\Lucas\Desktop\BA\Index").toPath());
         // configuration is set
         IndexWriterConfig config = new IndexWriterConfig(new StandardAnalyzer());
         // IndexWriter gets instantiated
         IndexWriter writer = new IndexWriter(dir, config);

         for (CrashLog crash : crashLogs) {
            ArrayList<String> crash_log_words = crash.getSetOfWords();
            FieldType fieldType = new FieldType(TextField.TYPE_STORED);
```

```
            fieldType.setStoreTermVectors(true);
            Document doc = new Document();
            for (String word : crash_log_words) {
               doc.add(new Field(LConstants.FIELD_NAME, word, fieldType));
            }
            doc.add(new Field(LConstants.FIELD_ID, crash.getPath(), fieldType));
            // document are added to the index using an IndexWriter object
            writer.addDocument(doc);
         }
            writer.close();
}
```

Listing 4.14: 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*. It is a base class for Directory implementations that store index files in the file system [12]. Next line of code, the analyser which is used for analysing the Lucene documents is defined. This can made using the Lucene class *IndexWriterConfig*, which holds all the configuration that is used to create an *IndexWriter*. TACL defines again a *StandardAnalyser*, 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 to store its vector of terms so that TF-IDF values can be computed for each word of each term of vector of each crash report. In this direction, TACL 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 term vector. Table 4.1 shows the structure of the hash map, taking as a reference the crash log presented in the listing 4.9.

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
	...	...

As shown in the table above, 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 of the term 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.
TACL computes tf-idf scores for all vector terms of all crash logs. Once this processed is completed, the similarity between the set of crash logs stored inside the hash map can be computed.

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, TACL computes cosine similarity between the previously generated vectors terms. The cosine similarity is just a measure of similarity between two vectors [39] (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, the vector that does not contain the term gets complemented with it by associating a tf-idf score of 0. In doing so furthermore, the two vectors have the same length so that their dot product can be computed. Figure below shows how the normalization process works.

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

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

As shown in the tables above, 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, TACL 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, TACL invokes the method *fillCrashLogBucket()*, 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, TACL 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, TACL 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, TACL 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".

The *BPMN*[2] diagram 4.4 explains and summarizes the above mentioned bucketing process provided by TACL.

At the end of the bucketing process, TACLsaves the bucket and print it out. The clustering process is now completed.

4.3 Linking

4.4 How to start TACL

First of all, a set of parameters and directories have to inserted in the static *Configuration Manager* file.

[2]Business Process Model and Notation

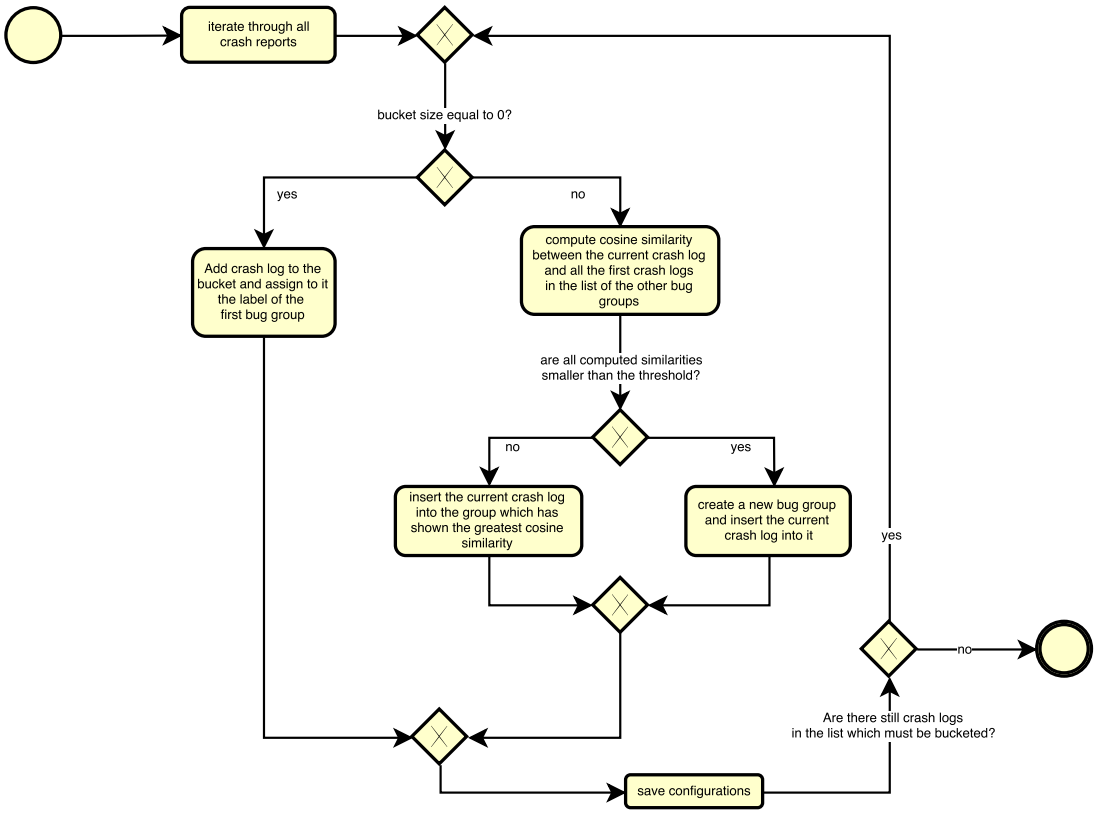

Figure 4.4: BPMN diagram describing the bucketing process

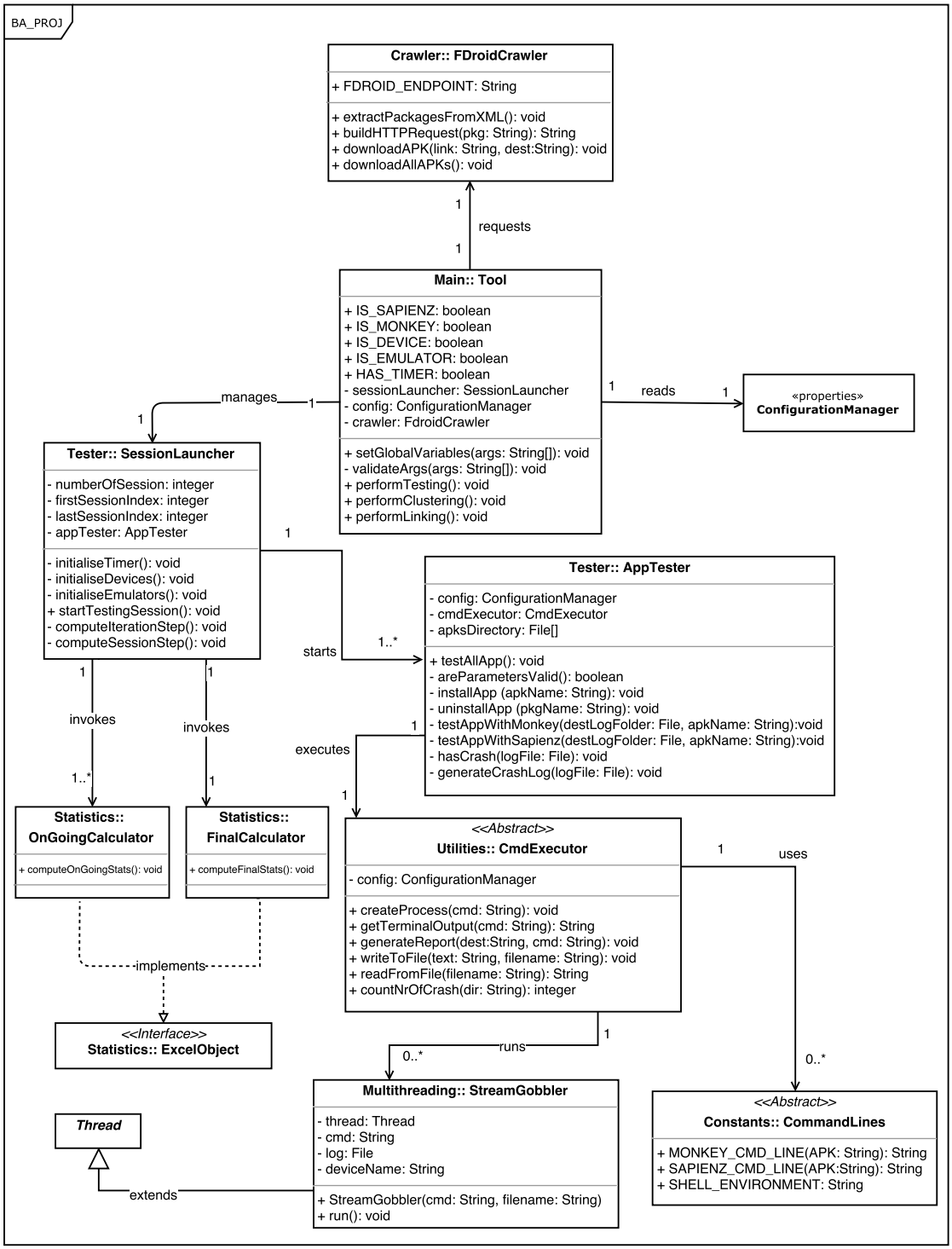

Figure 4.5: Class Diagram of the testing part of the tool

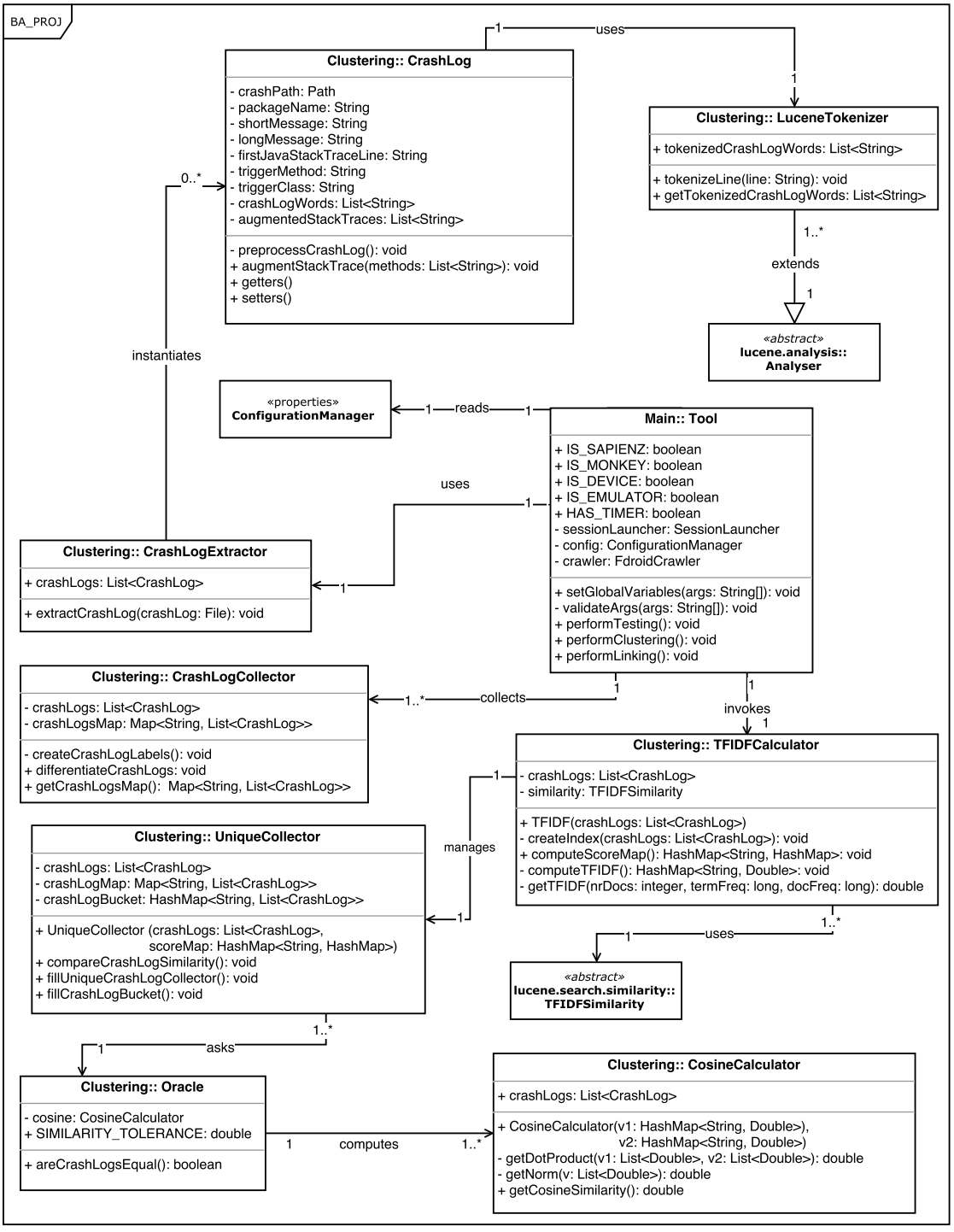

Figure 4.6: Class Diagram of the clustering part of the tool

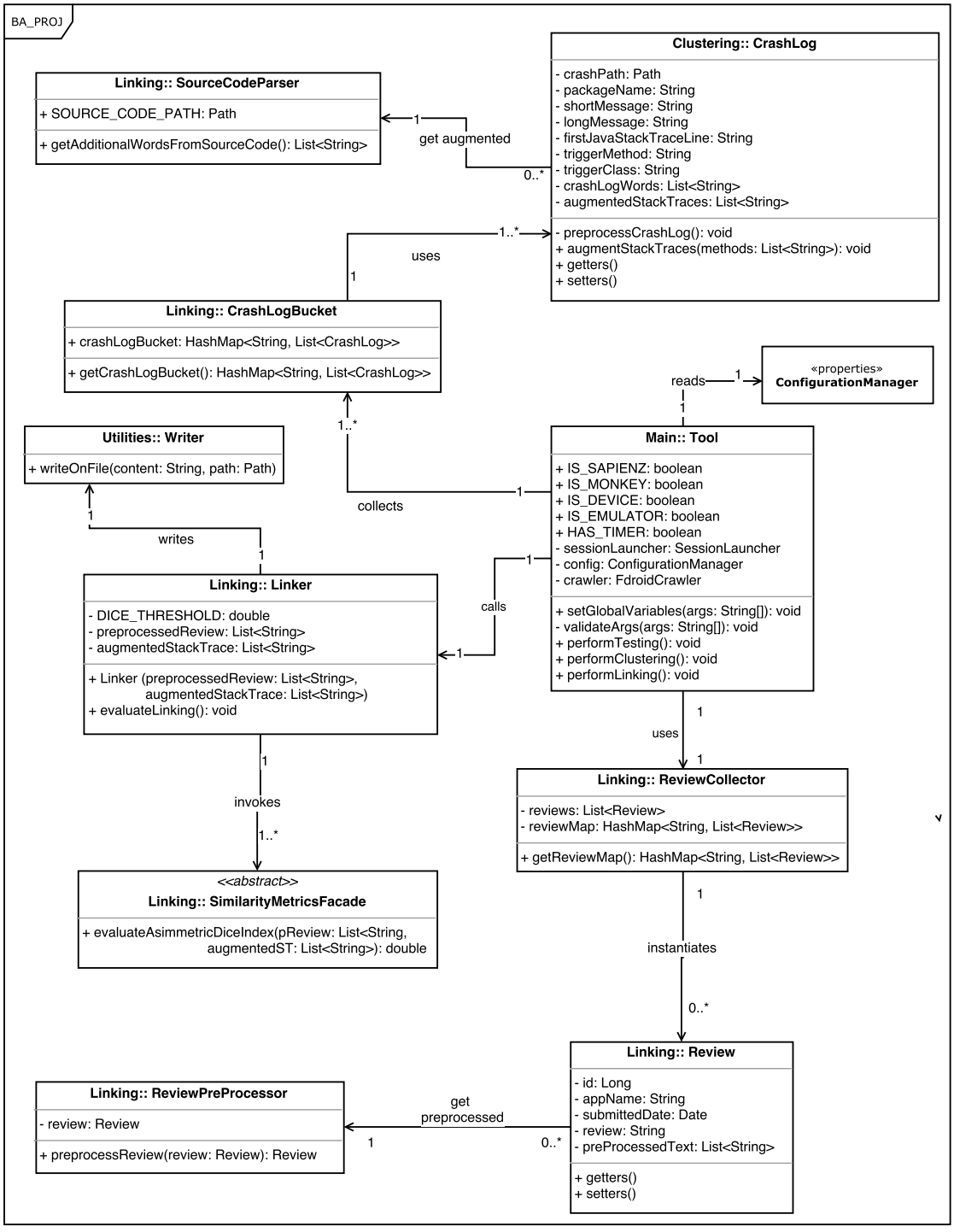

Figure 4.7: Class Diagram of the linking part of the tool

Chapter 5

Results and Discussion

Chapter 6

Conclusions and Future Work

Bibliography

[1] What is monkey testing? types, 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] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., 1990.

[4] 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.

[5] 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.

[6] W. Choi. Swifthand. .

[7] 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.

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

[9] W. E. W. Dijkstra. Edsger w. dijkstra. .

[10] A. S. Foundation. Apache lucene core. .

[11] A. S. Foundation. Class field. .

[12] A. S. Foundation. Class fsdirectory. .

[13] A. S. Foundation. Class textfield. .

[14] A. S. Foundation. Class tfidfsimilarity. .

[15] A. S. Foundation. Field termvector. .

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

[17] Google. Android monkey. .

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] M. D. Network. Testing process. .

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

[33] 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.

[34] 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.

[35] Statista. Number of apps available in leading app stores as of march 2017. , Mar. 2017.

[36] O. Team. Class runtime, oracle official documentation. .

[37] Wikipedia. Apache poi. .

[38] Wikipedia. black-box testing. .

[39] Wikipedia. Cosine similarity. .

[40] Wikipedia. Spiral model. .

[41] Wikipedia. tf-idf. .

[42] Wikipedia. V-model. .

[43] Wikipedia. Waterfall model. .

[44] Wikipedia. white-box testing. .