Bachelor Thesis

June 9, 2017

GUI usability and testing of mobile applications

Example subtitle

Lucas Pelloni

of 18.03.1993, 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

Testing is the action of inspecting the behaviour of a program, with the intention of finding anomalies or errors [21]. The goal behind Software testing is to reach the highest testing coverage finding the largest number of errors with the smallest number of *test cases* (a set of test inputs associated with an expected result when they are processed by a program).

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 [8] 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 [10] and big and complex applications might have a very large number of potential test scenarios, many of which are really difficult to be predicted. *Dijkstra* [7] 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 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 [8].

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 [24, 28]:

- 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* [10], 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 is applied 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* [26], the *Waterfall model* [27] and the *Spiral model* [25].
As shown, there exist few levels, 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 [10], in order to increase the reliability of the system. However as mentioned before, the process of finding the correct test scenarios, execute them, report their results and compare them to the previously written expected results 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.

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, [19], 149 billions downloaded only in 2016 [23] and 12 million mobile app developers (expected to reach up 14 million in 2020) maintaining them [6].
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 [12]. Furthermore, they are context-aware [19]. 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 [19].
For this reason, mobile applications require new specialized and different testing techniques [19] 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. If it would processed in a manual manner, it may be very time-consuming and may require a lot of programming. 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 [12, 14, 16, 19].

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 [20]. 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, etc.). For this reason, sometimes a time-consuming manual approach can be needed for testing an application [20].
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. Therefore, another different strategy is to incorporate opinions and reviews of the end-users during the software's evolution process.
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 [22].

test

1.1 Context

Therefore, since the growing competition characterizing mobile application marketplaces, like Google Play and the Apple App Store, ensures that only high quality apps stay on the market and gain users, developers have to pay particular attention to the quality of the apps they are developing and maintaining through adequate software testing activities. [14, 17].

D

1.2 Motivation

1.3 Motivation Example

1.4 Research questions

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.* [17].

2.1 Automated tools for Android Testing

Unlike traditional software, mobile applications are mainly exercised by user inputs.
In the mobile world, an extremely valid approach to ensure the realiability of these applications is the GUI[1] Testing.
In particular, in this kind of testing, each test case is designed and run in the form of sequences of GUI interaction events.
Depending on their exploration strategy, there are in general three approaches for creating a generation of user inputs on a mobile device [5, 14]: *random testing* [9, 14], *systematic testing* [15] and *model-based testing* [2, 4, 13].

Fuzz testing

When test automation does occur, it typically relies on Google's Android *Monkey* command-line [9]. Since it comes directly integrated in Android Studio, the standard IDE for Android Development, it is regarded as the current state-of-practice [**?**].
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*[2] [9].
The effort required for using *Monkey* is very low [5]. 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 [9].
The kind of testing implemented by Monkey follows a black-box approach. Despite the robustness, the user friendliness [5, 14] 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 [14].
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 *Monkey* [1].

[1] Graphical User Interface
[2] Application Under Test

Dynodroid [14] is also a random-based testing approach. However, this tool has been discovered being more efficient than *Monkey* in the exploration process [5].
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) [5].
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 [14]. All these events represent *non-UI events* and they are often unpredictable and therefore difficult to simulate in a suitable context (cita?).
Dynodroid views the *AUT* as an event-driven program and follows a cyclical mechanism, also known as the *observe-select-execute* cycle [14]. 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 [5, 14] and finally *executes* it. After the execution of that event it reaches a new state and can start the cycle 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 [14].

Systematic testing

The tools using a systematic explorations strategy rely on more sophisticated techniques, such as symbolic execution and evolutionary algorithms [5].
Sapienz [16] introduced a Pareto multi-objective search-based technique to simultaneously maximize coverage and fault revelation, while minimizing the sequence lengths.
It combines the above mentioned random-based approach with a new systematic exploration and as mentioned in the experimental results published on [16], *Sapienz* is an outperformer in the automated mobile testing area.
Indeed, in an empirical study described on [16], *Sapienz* has illustrated the strength of its approach. It found from a set of 68 benchmark apps, 104 unique crashes (while *Monkey* 41 and *Dynodroid* 13).

Model-based testing

Model-based tools for testing Android applications are quite popular [16]. Most of these tools [**?**, 2, 4, 13, 18] generate UI events from models, which are either manually designed or created from XML configuration files [16].
For example, *SwiftHand*[3] uses a machine learning algorithm to learn a model of the current *AUT*. This final state machine model [5] 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 [**?**]. *SwiftHand*, in a similar way to *Monkey* generates only touching and scrolling UI events and is not able to generate System events [5].

2.2 Usage of users reviews in Software maintenance activities

The concept of app store mining was first introduced by *Harman et al.* [11]. In this context, many researchers focused on the analysis of user reviews to support the maintenance and evolution of

[3]https://github.com/wtchoi/SwiftHand

mobile applications [17].

Chapter 3

Approach

Chapter 4

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. *IEEE Software*, 32(5):53–59, 2015.

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

[4] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices*, volume 48, pages 623–640. ACM, 2013.

[5] 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.

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

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

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

[9] Google. Android monkey. .

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] Wikipedia. black-box testing. .

[25] Wikipedia. Spiral model. .

[26] Wikipedia. V-model. .

[27] Wikipedia. Waterfall model. .

[28] Wikipedia. white-box testing. .