Bachelor Thesis

May 28, 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

# Abstract

# Zusammenfassung

# Contents

# List of Figures



# List of Tables

# List of Listings

# Chapter 1

# Introduction

## 1.1 Context

Software testing is widely recognised as an essential part of any software development process, presenting however an extremely expensive activity; in fact, its cost has been estimated at being at least half of the entire development cost [4].
While mobile applications are becoming so widely adopted, it is still unclear if they deserve any specific testing approach for their verification and validation [2].
Since, unlike traditional software, applications are mainly exercised by user inputs, an extremely valid approach to ensure the reliability 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.
The most famous automated GUI testing tools and their properties are discussed in the chapter *Related Work*.
Despite a strong evidence for automated testng approaches in verifying GUI application and revealing bugs, these state-of-art tools cannot always achieve a high code coverage [18]. 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 [18].
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 [19].

[1] Graphical User Interface

## 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.* [15].

## 2.1 Automated tools for Android Testing

Unlike traditional software, mobile applications are mainly exercised by user inputs.
In the mobile world, an exremely 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 [7, 11]: *random testing* [8, 11], *systematic testing* [13] and *model-based testing* [3, 5, 10].

### Fuzz testing

When test automation does occur, it typically relies on Google's Android *Monkey* command-line [8]. Since it comes directly integrated in Android Studio, the standard IDE for Android Development, it is regarded as the current state-of-practice [12].
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] [8].
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 [8].
The kind of testing implemented by Monkey follows a black-box approach. Despite the robustness, the user friendliness [7, 11] 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 [11].
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** [11] 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 [11]. 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 [11]. 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, 11] 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 [11].

### Systematic testing

The tools using a systematic explorations strategy rely on more sophisticated techniques, such as symbolic execution and evolutionary algorithms [7].
**Sapienz** [14] 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 [14], *Sapienz* is an outperformer in the automated mobile testing area.
Indeed, in an empirical study described on [14], *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 [14]. Most of these tools [3, 5, 6, 10, 17] generate UI events from models, which are either manually designed or created from XML configuration files [14].
For example, *SwiftHand*[3] 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.* [9]. 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 [16].

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, and B. Robbins. Testing Android Mobile Applications: Challenges, Strategies, and Approaches. *Advances in Computers*, 2013.

[3] 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.

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

[5] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. *SIGPLAN Not.*, 48(10):623–640, Oct. 2013.

[6] 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.

[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] Google. Android monkey. .

[9] 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.

[10] 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.

[11] 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.

[12] R. Mahmood and S. Malek. EvoDroid : Segmented Evolutionary Testing of Android Apps. *Fse*, pages 599–609, 2014.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.